TASK: UU03
CDRL: 05156
19 February 1993

Reuse Library Framework Version 4.1 Modeler Manual

Informal Technical Data

STARS-UC-05156/011/00
19 February 1993

TASK: UU03
CDRL: 05156
February 19, 1993

RLF Modeler's Manual
For The
SOFTWARE TECHNOLOGY FOR ADAPTABLE, RELIABLE SYSTEMS (STARS)

Reuse Library Framework
Version 4.1
SunOS Implementation

STARS-UC-05156/011/00
February 19, 1993

Data Type: A005, Informal Technical Data

CONTRACT NO. F19628-88-D-0031
Delivery Order 0000

Prepared for:

Electronic Systems Center
Air Force Systems Command, USAF
Hanscom AFB, MA 01731-5000

Prepared by:

Paramax Systems Corporation
Electronic Systems-Valley Forge Engineering Center
70 E. Swedesford Rd.
Paoli, PA 19301
under contract to
Paramax Systems Corporation
12010 Sunrise Valley Drive
Reston, VA 22091

TASK: UU03
CDRL: 05156
February 19, 1993

Data ID: STARS-UC-05156/011/00

Distribution Statement "A"
per DoD Directive 5230.24
Authorized for public release; Distribution is unlimited.

Developed by: Paramax Systems Corporation
Electronic Systems-Valley Forge Engineering Center
70 E. Swedesford Rd.
Paoli, PA 19301 under contract to
Paramax Systems Corporation

This software, developed under the Software Technology for Adaptable, Reliable Systems (STARS) program, is approved for release under Distribution "A" of the Scientific and Technical Information Program Classification Scheme (DoD Directive 5230.24) unless otherwise indicated. Sponsored by the U.S. Defense Advanced Research Projects Agency (DARPA) under contract F19628-88-D-0031, the STARS program is supported by the military services, SEI, and MITRE, with the U.S. Air Force as the executive contracting agent.

TASK: UU03
CDRL: 05156
February 19, 1993

RLF Modeler's Manual
Reuse Library Framework
Version 4.1
SunOS Implementation

Principal Author(s):

Timothy M. Schreyer *Date*

Approvals:

Task Manager *Richard E. Creps* *Date*

(Signatures on File)

TASK: UU03
CDRL: 05156
February 19, 1993

RLF Modeler's Manual
Reuse Library Framework
Version 4.1
SunOS Implementation

Change Record:

Data ID	*Description of Change*	*Date*	*Approval*
STARS-UC-05156/003/00	Original Issue	November 1992	*on file*
STARS-UC-05156/011/00	Updates for version 4.1	February 1993	*on file*

REPORT DOCUMENTATION PAGE

Form Approved
OMB No 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
		Informal Technical Report

4. TITLE AND SUBTITLE

RLF Modeler's Manual

5. FUNDING NUMBERS

F19628-88-D-0031

6. AUTHOR(S)

Paramax Corporation

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Paramax Corporation
1210 Sunrise Valley Drive
Reston, VA 22090

8. PERFORMING ORGANIZATION REPORT NUMBER

STARS-UC-05156/011/00

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Department of the Air Force
Headquarter, Electronic Systems
Hanscom AFB, MA 01731-5000

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

05156

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION / AVAILABILITY STATEMENT

Distribution "A"

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

This manual is intended for the library domain modeler of a reuse library hosted on the Reuse Library Framework (RLF). Some information on RLF modeling maybe of interest to a library administrator managing the day-to-day operations of an RLF reuse library and making improvements. Specific information on operating an RLF library is given in the **RLF Administrator's Manual.**

14. SUBJECT TERMS

15. NUMBER OF PAGES
108

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unlcassified	Unclassified	Unclassified	SAR

Contents

List of Figures

List of Tables

1 Introduction

1.1 Scope

This manual is intended for the library domain modeler of a reuse library hosted on the Reuse Library Framework (RLF). Some information on RLF modeling maybe of interest to a library administrator managing the day-to-day operations of an RLF reuse library and making improvements. Specific information on operating an RLF library is given in the **RLF Administrator's Manual.**

The library domain modeler is the person responsible for creating the knowledge model which provides the structure for the reusable assets in an RLF reuse library. This role includes examining the areas common to the reusable assets in the library (the "domain"), modeling and encoding the domain when it has been identified, and making any changes to the library domain model as the library evolves to include more assets or better accommodate the library user. Requests to change the library domain model may often come from the library administrator.

This manual also addresses the modeling and encoding of library domain advice. The library domain modeler also models and encodes the advice modules which are attached to the library model and aid the library user in the identification and location of the correct reusable asset. This role includes examining the library advice domain, modeling and encoding advice modules, or "inferencers," and updating these inferencers to reflect changes or improvements in the reuse library or its structure.

This manual assumes the modeler has a basic understanding of the UNIX operating system. If the modeler intends to add to or modify the set of built-in actions calling Ada procedures, it assumes a good understanding of the Ada language and the SunAda or Verdix Ada compilation system. This version of RLF can support execution using Emeraude PCTE v12.3 as an underlying object management system. If RLF is run with PCTE, it is assumed that the user understands PCTE and the Emeraude product, including the ability to construct *esh* scripts.

1.2 Identification

The **RLF Modeler's Manual** provides the information necessary for an RLF reuse library domain modeler to model, encode, and build an RLF reuse library specification and the library itself. It also defines how to model, encode, and install the RLF library advice modules called "inferencers." More detailed information of managing the reuse library after it has been constructed can be found in the **RLF Administrator's Manual.** If RLF has been constructed to run with PCTE, then certain guidelines must be followed when modeling for PCTE. PCTE-specific information has been gathered in appendix E.

1.3 Product Overview and Rationale

The Reuse Library Framework (RLF) is a knowledge-based system for reuse library construction and operation. By structuring a set of reusable software assets in a knowledge network and representing their descriptions and interrelationships in the network, the RLF increases the library user's chances of finding and extracting the reusable asset that is desired. The knowledge-based representation also helps enhance the user's understanding of the system from which the reusable assets are taken,

and allows an "intelligent" help mechanism to aid the user with retrieval of assets.

The remainder of this document is organized as follows. Section 2 describes RLF's approach to structuring a reuse library through the use of domain models and also discusses the domain of a library's inferencers. Section 3 discusses the fundamentals of the AdaKNET knowledge network subsystem which represents the library domain model. Section 4 describes the Library Model Definition Language (LMDL) which is an application-specific language which is used to encode the library domain model. Section 5 presents the AdaTAU rule-based inferencing subsystem which is used by RLF to host the reuse library advice mechanism. Section 6 describes the Rule Base Definition Language (RBDL) which is an application-specific language used to encode the library advice domain into reuse library inferencers. The next section, section 7, details the use of the RLF applications `Lmdl` and `Rbdl` which translate specifications in LMDL and RBDL into persistent representations of the entities which make up an RLF reuse library. Sections 8 and 9 give guidelines and hints for developing knowledge models in LMDL and RBDL and testing them. The next section, section 10, shows how to make changes to existing knowledge models in LMDL and RBDL, and section 11 describes how to report problems with any of the modeling subsystems. Finally the appendices contain summaries of the LMDL and RBDL language syntaxes, a starter LMDL library model specification including an example action sub-model, the syntax of the `.rlfrc` start-up file and an example, a discussion of PCTE issues when using RLF and PCTE, and a short presentation of a SNDL (pre-RLF 4.0 modeling language) to LMDL translator.

1.4 Notation Used in This Manual

Several different typefaces are used in this manual to notate objects of different kinds. The names of manuals are printed in a **bold** typeface. The names of UNIX tools or utilities are printed in *italics.* The names of directories and files, the text of UNIX shell scripts, environment variable names, and the names of RLF applications are printed in `typewriter` typeface. Examples of Library Model Description Languages (LMDL) and Rule Base Definition Language (RBDL) also appear in `typewriter` typeface.

2 RLF Domain Model Approach

2.1 Purpose of Domain Model

RLF's approach to managing a reuse library is based on the principle that a highly-structured reuse library will be easier to browse and understand. The structure of an RLF library is provided by a knowledge network which not only classifies the assets in the library in a hierarchy from general to most specific, but also describes the relationships between assets and the part they may play in the composition of a larger system. Reuse libraries are most effective when they contain assets from a relatively small subgroup of all possible assets and when the assets share some common traits such as the subsystem they come from. This way it is easier to understand the connections between assets and their role in larger collections of assets such as a software subsystem.

When an RLF library model is constructed to structure the assets which will be in the reuse library, the first step is to identify the area common to all the assets which will be in the reuse library. This area is called a "domain." The process of defining the domain is called "domain analysis," and the process of encoding that domain into some sort of structure is called "domain modeling."

These activities in general can become very complex and it is beyond the scope of this manual to fully describe them here. This manual will give a sketch of the domain modeling process in order to give RLF library modelers a start in creating RLF reuse libraries.

The "domain model" is the final product of domain modeling. By capturing the domain model of the reusable assets in the library as an RLF knowledge network, the level of understanding of the assets in the library increases significantly. This in turn improves the chances that an asset extracted from the RLF library will be immediately useful to the library user. The key to effective reuse is to minimize the time taken to find and retrieve the asset to be reused, and to increase the chances that the asset can be reused without much alteration. The domain model approach ensures that there is enough information in the library structure to meet these goals.

2.2 Domain Identification

Before a library domain model in encoded in the RLF knowledge definition languages, LMDL and RBDL (described in sections 4 and 6, respectively), the actual extent of the domain must be identified. In this process, the collection of reusable assets to be available in the reuse library are examined to find the traits that they have in common. This might be as little as coming from the same software subsystem to as much as the assets sharing large amounts of data or having complex interdependencies. The assets might share the fact that they are all reusable implementations of common abstract data types in a certain language or are all reports about the performance of different software packages on different projects throughout the company. By narrowing what assets and traits fall within the perceived domain of the reuse library, a tighter and more comprehensive domain model for the library can be developed. A tight domain model ensures that the library will be more easily understood, and that leads to more effective use of the library.

When identifying the domain of the library being developed, it is also important to consider future additions to the library and not just the reusable assets that will populate the library initially. A good reuse library should be able to accommodate the addition of new or better reusable assets from the library's domain without an excess of library modifications or restructuring. Although restructuring may need to take place to make the library more usable in response to users' needs, the library domain should be broad enough to allow easy inclusion of new assets. When identifying the domain, a balance must be reached between creating a very tight domain which enhances understanding and leaving the domain flexible enough to evolve easily as the library matures.

If in the process of examining the initial assets in the library, several distinct domains are identified, it is often better to establish two or more libraries to focus on each of these domains. Libraries should be of comprehensible size; a library which is too large frustrates users and means the library will not be used and will be ineffective. On the other hand, libraries which are too small often overlook the connections between the assets in the library and other objects in the environment where the assets are used. Although it may be easiest to identify the largest domain possible to include all available reusable assets, this often leads to ineffective libraries and the modeler should not be afraid to develop several libraries with smaller domains if that seems appropriate.

2.3 The Library Model and Library Advice Domains

Although an RLF reuse library should only support one library domain for maximum effectiveness, the knowledge in the library's domain must be necessarily split because of the way RLF supports the modeling of the domain knowledge. This split has a natural separation and is realized in RLF as the difference between the "library model" domain which is encoded in the Library Model Definition Language (section 4) and the "library advice" domain which is encoded in the Rule Base Definition Language (section 6).

The library model domain describes the relatively "static" associations between elements of the domain. This includes the relationships assets have with other assets and descriptions of the assets. The library model domain for a software subsystem, for example, includes descriptions of all the major parts of a subsystem, the minor parts which compose the major parts, the parts that compose the minor parts, etc. and the dependencies and connections between them. The library model domain for the subsystem also contains information describing the parts like performance, size, origin, host-system requirements, version number, etc. All of this information about the library's domain can be determined by inspecting and researching the assets that will be in the library. The library model domain is the chief structuring element of an RLF reuse library and is represented using the AdaKNET semantic network subsystem (see section 3).

The library advice domain describes the relatively "dynamic" information about the user's search for a particular asset in the library. The library advice domain is used to model the advice-giving capability of RLF reuse libraries. The library advice domain includes much of the same information on structure and dependencies as the library model domain but also includes information about a library user's particular requirements when searching for a reusable asset. This information can include such things as the user's target hardware, minimum performance requirements, space constraints, minimum verification levels, languages preferences, and level of expertise in the library's domain, among other things. Although much of the subject of the library advice domain can be taken from the library model domain, it is important to capture the other parts of this domain in order to build useful advice-giving mechanisms for the reuse library. The library advice domain is represented using the AdaTAU rule-based inferencing system (see section 5).

2.4 Narrowing the Domain

First efforts at identifying a domain that encompasses a collection of reusable assets will often produce a very broad and general domain. This very loose domain is not a good foundation for an RLF reuse library for some of the reasons mentioned above. This general domain needs to be "narrowed" so that it contains the pertinent information about the assets in the library and excludes extraneous information which would not aid a user in deciding which reusable asset is the one best suited to the task. Deciding which information is pertinent and which is extraneous can be a difficult task and this may change as the library matures and the library's users requirements change.

It is useful to keep in mind the potential library user's perspective while narrowing the library domain. Information about the library's assets which is important to users should play a central role. The user will be browsing and searching the library with a certain goal asset in mind and will have classified the asset mentally before searching the library for it. If the modeler can predict what criteria the user will be using when searching for assets in the library and narrow the library

domain accordingly by including this key information, then the library will be more highly usable and useful.

When narrowing the domain to fit the users, it is also beneficial to determine which parts of the domain fit better as "static" descriptions of the assets in the library model, and which parts are more of the "dynamic" per-user information which fit better into the library advice domain. A user should be able to find out all the interesting information about an asset from browsing the library model domain, but the library advice domain should include the domain information about the user's perspective and motivation for browsing the library.

2.5 Gathering Domain Information

Several sources can be consulted to gather the information about the domain of the reuse library. One primary source is the reusable assets themselves. By examining and researching the reusable assets, many of the interdependencies and qualities of the assets can be determined. For example, if a library was being built for a collection of software modules, the source code itself could be examined to obtain dependencies between modules, the modules could be built and executed to obtain performance information and memory requirements, or documentation in the modules could be used to help describe them in the library model.

Another source of useful domain information are personnel who work in the domain of the library or work with the assets that will be in the library. These people can usually provide information about overall structure and importance that is not evident in the assets themselves, especially if the assets are coarse-grained parts of a large system. This larger domain knowledge can provide the top-level structure of the domain model and is what makes RLF library domain models useful as understanding tools for the library's domain. Conducting interviews of these experts in the library's domain and asking questions about the domain is the usual way this information is gathered.

Similarly, interviews of prospective users of the reuse library (if they differ from the experts in the domain mentioned above) can provide information about how the library should be organized for maximum effectiveness. Although users may not be able to lend much information about the parts that they wish to extract from the library, they provide a lot of information about what is important when the library is searched, and encoding some of this search criteria into the library model domain, and especially the library advice domain, can greatly increase the effectiveness of the reuse library.

Most of the information which will end up in the library model domain will be gathered by examining and researching the assets which will be in the library initially, researching materials on the domain itself, and interviewing experts in the library's domain. Most information that is useful in the library advice domain will come from the assets themselves and from a determination of the needs and perspective of the library's users. A lot of the library user's perspective can often be gathered from interviews or questionnaires conducted with potential users of the reuse library being developed.

2.6 Gathering Domain Assets

It is sometimes beneficial to gather as many assets in the library's domain as possible before modeling the domain. This gives insights about the domain which may not be apparent from any one collection of assets in the domain. An RLF library can be modeled and established without any domain assets to start with, but this will require a lot of guesswork about what the meaningful traits of these assets are that should be included in the domain model. When the domain model is actually populated with assets, the model may need to be reworked to address qualities of the assets that were overlooked or given too much or too little emphasis in the initial model.

It is probably easiest and most effective to develop the RLF library domain model with the initial set of reusable assets which will be in the library on hand. Conducting a search to find some additional assets in the domain is also an effective way to ensure that the fielded library will be able to accommodate additional assets in the domain without too much restructuring. This flexibility is important because reuse libraries that can evolve in response to the requirements of their users are likely to be more effective.

2.7 Adding Initial Domain Assets

When the reuse library domain is being identified, narrowed, and refined it is good to do this while keeping in mind where each asset in the library will be situated. Each asset should be thought of as one example, or "instance," of some general class or category of things described in the model. Examining the assets that will be in the library, the modeler can usually envision a very similar asset that fits the general description of the first asset but is slightly different. The model should be established so that if this asset was located later, it could fit easily into the model with limited modification. It would become another instance of the category or class. In this way, each initial asset that was used to help define the library domain model in the first place would fit tightly into the model while still leaving enough generality in the model to accommodate new assets added later.

2.8 Encoding the Library Model and Library Advice Domains

Once the domain of the library being developed has been identified, and then narrowed and refined to neatly describe the domain of the library and the assets the library will contain, the model needs to be encoded in the RLF knowledge definition languages so that the library can be created. The library model domain which provides the structure for the library is specified in the Library Model Definition Language (LMDL) (section 4) and represented using the AdaKNET semantic network subsystem described in section 3. Hints on encoding the library model domain with LMDL are given in section 8. The library advice domain is encoded using the Rule Base Definition Language (RBDL) (section 6) and represented using the AdaTAU rule-based inferencing subsystem described in section 5. Hints for encoding and testing the library's advice modules, or "inferencers," are given in section 9. Section 7 gives information on how to turn LMDL and RBDL specifications into an RLF reuse library using the LMDL and RBDL language translators.

3 AdaKNET Library Models

3.1 Overview

This section discusses the AdaKNET semantic network representation used to encode the hierarchical structure of RLF reuse libraries. It presents a brief introduction to the semantic network and its role in the reuse library and then discusses the entities which compose a library model represented with AdaKNET and the semantic rules which are imposed on those entities to ensure a consistent knowledge representation.

3.1.1 Semantic Network Subsystem

AdaKNET is a knowledge representation formalism based on KL-ONE [BS85]. Other examples of representation systems in this family are NIKL [KBR86] and KNET [FHM+83]. Members of this family of representation systems are often called "semantic networks." AdaKNET provides its user the ability to describe a domain by creating a model of that domain with AdaKNET's structure enforcing certain consistencies between the components of that model. Instances of AdaKNET can be thought of as complex graphs of classes of things and the relationships between those classes. An instance of AdaKNET also contains members of the classes it describes and their relationships with other classes and members of classes.

The AdaKNET implementation supports strict specialization (subsumption) semantics, range and value constraints on the relationships of a category or object, single and multiple inheritance of relationships and actions, and the subdivision of relationships. The implementation also distinguishes generic classes of things (categories) from instances of these classes (objects). Files, strings, and integers can be associated with categories and objects. In addition, AdaKNET supplies a general action mechanism which allows inheritable, constrainable actions to be declared between classes. These features provide sufficient modeling generality to describe a broad spectrum of domain models for building reuse libraries.

3.1.2 The Multiple Roles of AdaKNET Models

The primary role of the AdaKNET model in an RLF reuse library is to describe and structure the reusable assets in the library. Assets are defined by their position in the model hierarchy and their relationships to other assets and other parts of the model. In this way, the AdaKNET library model "classifies" the assets of the library. Some of the advantages of using a semantic network subsystem like AdaKNET to build the library model are that the descriptions of the assets are themselves described elsewhere in the library model and the assets' relationships and dependencies on other assets are easily discovered.

Another role that the AdaKNET library model plays stems from the fact that it is a "domain model." Since the assets in the library are classified according to the role they play within the domain they come from, a well-developed model of the domain is automatically present and accessible from RLF applications like the `Graphical_Browser` and `Library_Manager`. Because of this, an RLF library can be used as an aid in learning the domain for which the library was created. For example, browsing the sort and search algorithms library model, one can gain an understanding of the important features of sort and search algorithms (or algorithms in general), how they

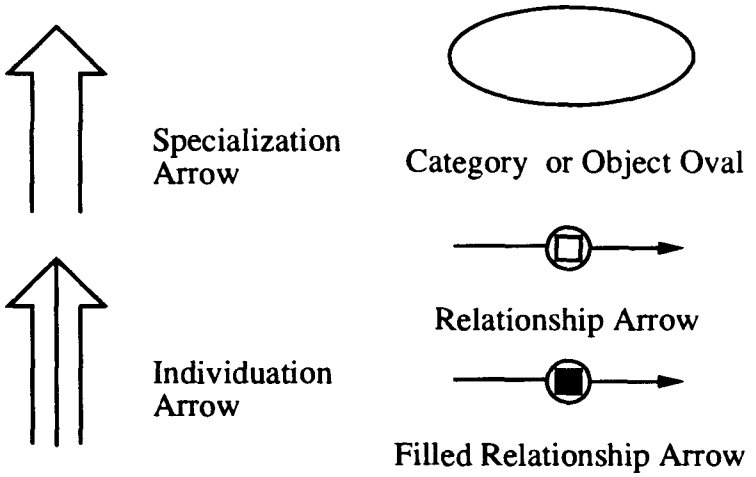

Figure 1: Graphical Notation for AdaKNET Knowledge Models

interact with other features, and how some of these algorithms have been implemented. Browsing the STARS Ada/Xt library educates the user in the architecture of Ada/Xt, how its functionality is packaged, and how the packages rely on each other. This library also describes all of the types in Ada/Xt.

A third role an AdaKNET model can have is to provide information about system composition and configuration. The model can include the description of an entire subsystem, and then describe the parts of the subsystem, and then the parts of the parts, etc. until all parts which make up the subsystem have been described. The model can also describe which parts are always required and which parts are needed when other parts are included in the composition of a subsystem.

3.2 Graphical Notation for Library Models

There is a standard graphical notation for AdaKNET knowledge models which is used throughout this manual. The basic components of this graphical notation are shown in figure 1. Explanations of the meanings of these components is given in the following sections.

3.3 AdaKNET Entities

The following section introduces and describes the fundamental entities which comprise AdaKNET models in RLF. It discusses the semantics of each entity and its interrelationship with the others. How the AdaKNET entities are encoded in the Library Model Definition Language (LMDL) is discussed in section 4.

3.3.1 Library Models

Library models are the highest level entity in RLF. They contain the definition of all the elements of any individual RLF reuse library. The underlying AdaKNET semantic network subsystem used to

represent the library model ensures that each library model is consistent throughout and conforms to the semantics described in this section.

Partitioning of large library models can be achieved using the "incremental" features of LMDL. This is further described in section 4. Library models can be partitioned as long as the partitions are built in a particular order and later models only *add* AdaKNET entities to earlier models. Each partition can replace its predecessor or be given a new name so that the partition being extended can be retained in its original state. Allowing the modeler to supplement the contents of one library model with more definitions supports, for example, the inclusion of whole topical sub-models when required or the simultaneous addition of all of a library model's objects. Large library models can be partitioned so that different versions of the same library can be presented to different users including people just interested in learning the domain or developers intent on extracting real reusable assets.

Small updates to large models can be made using the incremental feature of LMDL to avoid the translation time of the large model specification. These updates should eventually be merged into the main library model, however, or propagation of many small incremental partitions can cause configuration management problems reproducing the library from all the library model specifications on hand.

Future versions of RLF will support even greater interaction of library models so that complete models can reference entities in other library models. This way libraries and library models could be easily composed from several libraries or library models. This promotes the reuse of the library model itself in addition to a library model's library assets. Model development can be an expensive process, and reuse of the results of previous modeling efforts is highly desirable.

3.3.2 Categories

In AdaKNET library models, the principal entities are "categories" and "objects" (section 3.3.3). A category models a class of things, such as the class of all algorithms or the class of all programming languages. An object represents one particular thing; for instance, **Quicksort** represents a specific **Algorithm**, and **Ada** represents a specific programming language. Thus, categories are roughly equivalent to "classes" in object-oriented systems, while objects are roughly equivalent to "instances." No two categories may have the same name. A hierarchy of categories provides the basic structure of every RLF reuse library.

Every library model has a special category called the "root category." The root category describes the most general thing in the library model. The root category is the very top of the hierarchy of categories in the library model with every other category directly or indirectly descended from it. Each library model may only have one root category.

Specialization

Categories in AdaKNET are organized into a specialization hierarchy. A category *A* "specializes" another category *B* if *A* represents a subset of the class described by *B*. Taken simply, this means that the most general category appears at the top of the hierarchy with more specific categories below it and the most specific categories at the very bottom.

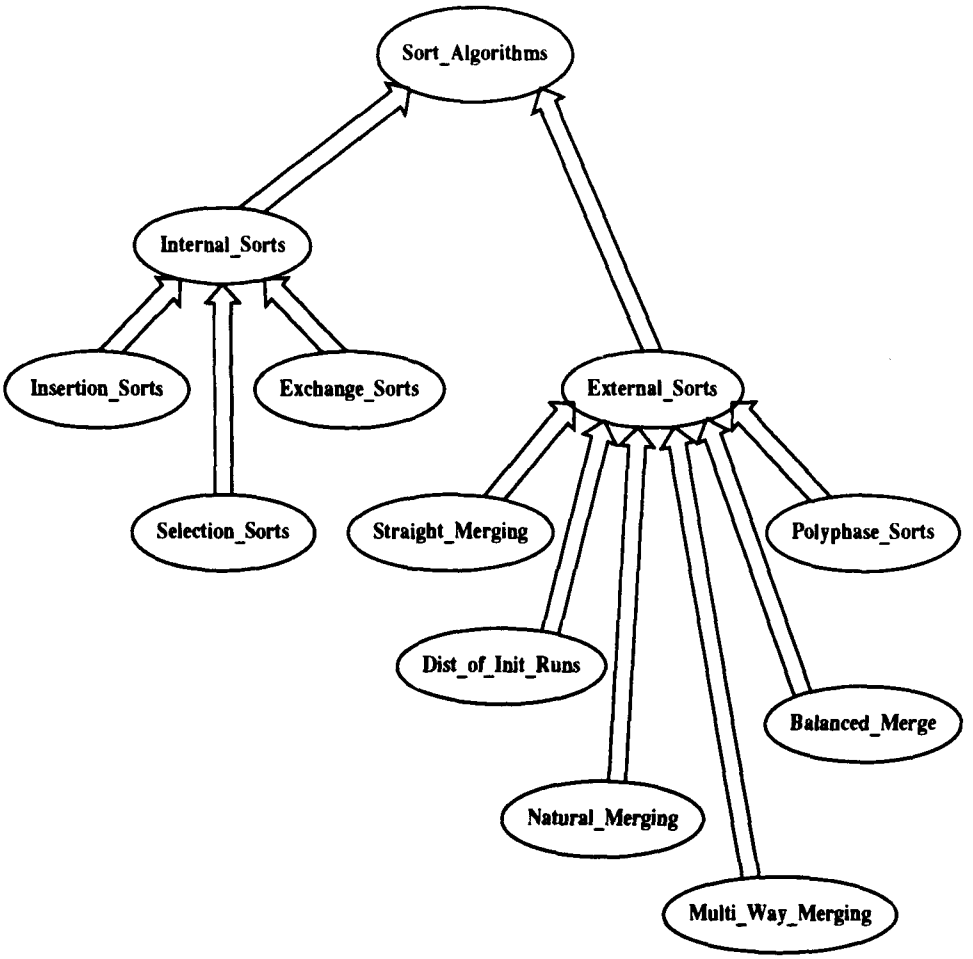

Figure 2: An Example of Specialization of Categories

A sample specialization hierarchy is shown in figure 2. This is a subsection of the sort and search algorithms example library delivered with RLF. We see that the category "`Internal Sorts`" is defined in terms of "`Sort Algorithms`", that is, "`Internal Sorts`" specializes "`Sort Algorithms`". Similarly, "`Selection Sorts`" specializes "`Internal Sorts`". Conversely, we say the category "`Sort Algorithms`" "subsumes" the category "`Internal Sorts`". The subsuming categories are called "supercategories" of the subsumed categories, and the subsumed categories are called "subcategories" of the subsuming categories. Because "`Internal Sorts`" is directly linked to "`Sort Algorithms`", we further say that "`Sort Algorithms`" and "`Internal Sorts`" are in a *parent/child* relationship.

Specialization and subsumption are acyclic and transitive relations. So, in figure 2, "`Selection Sorts`" are kinds of "`Sort Algorithms`", as well as kinds of "`Internal Sorts`". Specialization and subsumption are also many-to-many relations, that is, a category may have multiple parents and children.

Inheritance

In AdaKNET, a child category "inherits" the relationships and actions of its supercategories; that is, each relationship or action of the supercategory is also a relationship of the subcategory. Relationships are fully described in section 4, and actions are described in section 3.3.6. Herein lies the power of specialization: to define a category, one only needs to specify a category's parents and that information which distinguishes the category from its parents. Such distinguishing information may be new relationships or actions introduced at the subcategory, or further restrictions on relationships or actions that are inherited. (Restrictions and subdivision of relationships called differentiation are discussed in a later section.) The semantics of the specialization relation are, essentially, that any object of the child category is also an object of the parent category. For this to be true, child categories can only strengthen the restrictions of inherited relationships. This notion of *subsumption preserving semantics* is central to understanding what constitutes legal AdaKNET models.

Multiple Inheritance

AdaKNET allows a category to specialize more than one supercategory. This capability, called "multiple inheritance," allows a category to inherit the relationships of all of its parents. When the parents have non-overlapping sets of relationships, multiple inheritance works in the same way as single inheritance. If some parents share a relationship which descends from a common ancestor (i.e. there exists a single category which subsumes the parents and from which the parents inherit the relationship), the relationship is inherited with the conjunction of the parents' restrictions on the relationship. This is discussed in more detail in section 3.3.4.

3.3.3 Objects

Library model "objects" are very important to the reuse library. Objects describe particular things — an individual member of a category. As such, they are the knowledge representation of the actual reusable assets in the reuse library. The assets themselves are bound to the objects which represent them by attributes (see section 3.3.5).

As the model of an individual member of a category, an object inherits the relationships and actions of its parent. Inherited relationships at objects are often called "particular relationships" since they describe an instance of the relationship for this particular object. This is one of two ways particular relationships can be introduced at object categories (differentiation is the other way). In fact, this is the only way that particular relationships are created. No new relationships may be introduced at an object; all relationships must correspond to a relationship of one of the subsuming categories.

Individuation

Each object in an AdaKNET domain model is an instance of some category, that is, it "individuates" one or more categories. Figure 3 illustrates individuation. Here **"Heap Ada"** individuates **Heapsort** and **Ada** individuates **"Source Language"**. Individuation is preserved by subsumption, so that **"Heap Ada"** also implicitly individuates **"Selection Sorts"**, and **"Sort Algorithms"**, and **Thing**. In cases where it is important to distinguish between explicit and implicit individuation, we will add the term "direct" or "indirect" to the description; e.g., **"Heap Ada"** directly individuates **Heapsort** and indirectly individuates **"Selection Sorts"**.

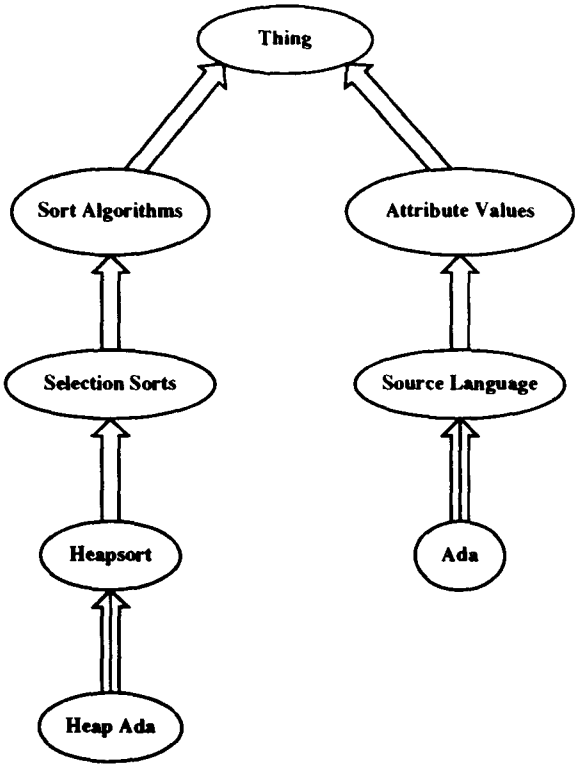

Figure 3: An Example of Individuation of Categories by Objects

Individuation, like specialization, is also a many-to-many relation; a category may be directly individuated by several objects, and an object may directly individuate several categories.

3.3.4 Relationships

"Relationships" describe the structure and qualities of categories. For instance, a human has components such as a head, a torso, arms, and legs, and has qualities such as height, weight, and gender. Such qualities are represented in AdaKNET by associating relationships with a category. For example, a category representing humans might include relationships for height, weight, eye-color, etc.

Relationships in AdaKNET domain models serve either of two purposes: to indicate the general *types* of things that satisfy a given quality or to specify the *exact* thing (the "filler", as described later) that instantiates ("satisfies", as described later) a given quality of an object. The distinction between these two is discussed in the following excerpt from [BS85]:

> This difference is motivated essentially by the "attributive/referential" distinction in the philosophy of language. Imagine a situation in which an alligator's tail has fallen off. We might remark, "The alligator's tail lay wriggling on the ground." Or, we might say something like, "Don't worry, the alligator's tail will grow back again." The "tails" talked about must be different in the two cases – in the first, we are referring to the previous filler, the actual piece of protoplasm that used to be the alligator's tail. In the second, because the alligator's tail will not reattach itself to the alligator, we must mean something else by "alligator's tail." We are in fact talking in a general way about anything that will eventually play the relationship of "tail" for the alligator.

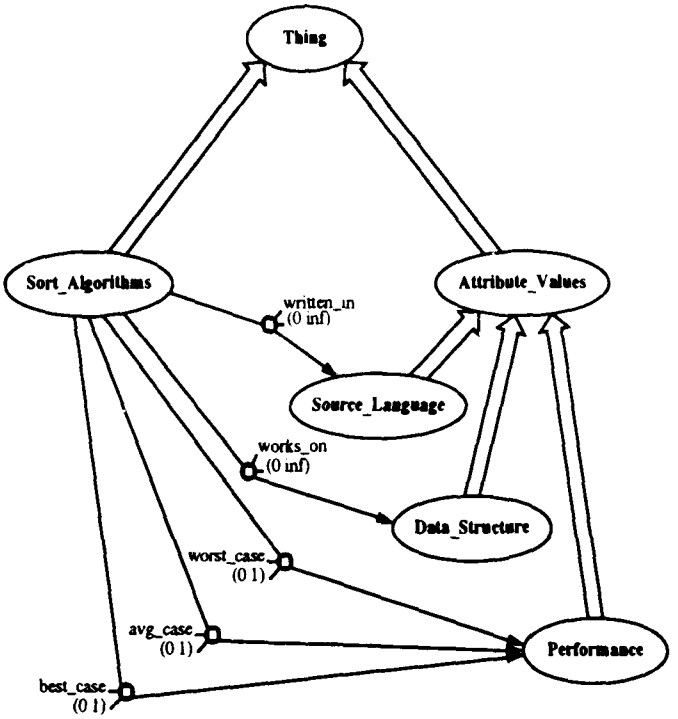

Figure 4: An Example of Relationships

The term "relationship" will be used to indicate the general, attributive flavor of relationships. A "particular relationship" will be used to indicate the specific, referential flavor of relationships and "filled relationship" will be used to describe a particular relationship which relates two objects (and not just an object and a category).

Relationships are templates that identify and describe what type of thing the relationship's fillers should be (e.g. the height of a human is a length) and how many fillers it should have (e.g. a human has up to two legs). Figure 4 illustrates the notion of relationships and the associated graphical conventions. In this figure, the category "**Sort Algorithms**" has five relationships which describe the qualities all algorithms share. For example, every algorithm has a worst case, average case, and best case performance (even though some may be the same in some cases).

Relationship Restriction

The "type" or "value restriction" of a relationship's fillers is specified by a category associated with the relationship. In figure 4, the relationship **works_on** has type "**Data Structure**", indicating that "**Sort Algorithms**" do work on "**Data Structure**"s. Objects which fill the **works_on** relationship must therefore be objects of type "**Data Structure**", or be objects of some subcategories (directly or indirectly) of type "**Data Structure**".

The cardinality of a relationship's fillers is specified by a relationship's "range restriction" (or, simply "range"). A range restriction consists of a lower and an upper bound on the number of fillers the relationship is allowed. If the lower and upper bounds of a relationship range are equal, we say the relationship has been "converged." Infinity, **inf**, as an upper bound indicates that an

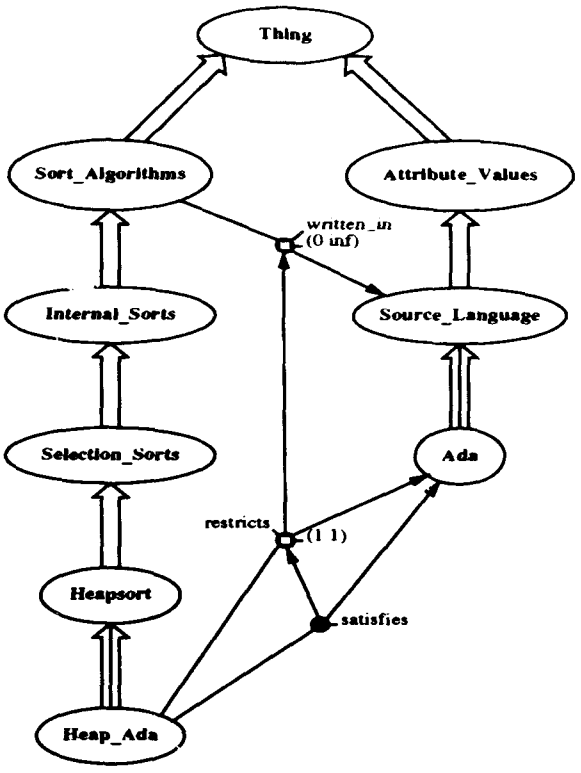

Figure 5: An Example of Relationship Restriction and Fillers

unlimited number of fillers are possible.

Relationship restriction is a mechanism whereby a category constrains the range and/or type of inherited relationships. All relationship restrictions must preserve the semantics of the specialization and individuation relations. Since types and ranges constitute *necessary* conditions on fillers, this means that these conditions may not be weakened by relationship restriction. Thus, one may restrict an inherited relationship's range to be a smaller interval than the range of the parent's relationship, and/or one may restrict an inherited relationship's type to be some specialization of the type of the parent's relationship. Relationship restriction is denoted via the "restricts" relation. Note that the restricts relation does not introduce a new relationship, but rather tightens the range or narrows the type of an inherited relationship.

Figure 5 is a combination of figures 3 and 4. It shows a relationship called `written_in` which indicates that "`Sort Algorithms`" can be written in any number of "`Source Language`"s. It also shows the `written_in` relationship inherited at the "`Heap Ada`" object and becoming a particular relationship. The particular relationship `written_in` is then restricted in type (or value) to always be `Ada` and restricted in range to exactly one. Because the lower and upper bounds of `written_in`'s range are equal, it is converged. Figure 5 also shows that the relationship has been filled by the filler `Ada`; `Ada` satisfies the relationship `written_in` for the object "`Heap Ada`".

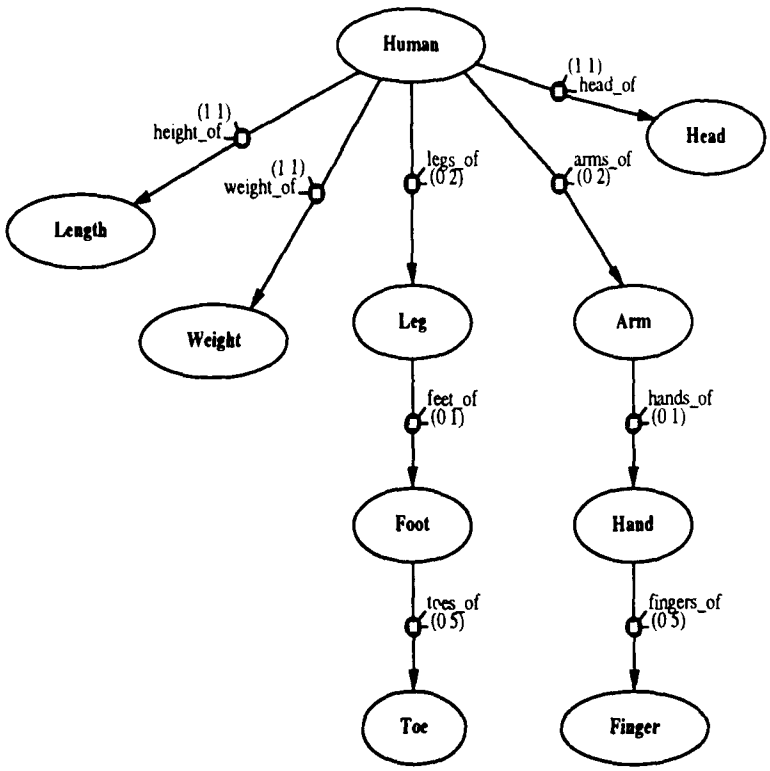

Figure 6: An Example of a Model's Aggregation Hierarchy

Aggregation

The "aggregation" relation is the relation that exists between two categories when a relationship describes them. The "aggregation hierarchy" is a hierarchy constructed by following relationships from one category to each type category of each relationship, then collecting the relationships of each type category and following them to their type categories, etc. until a transitive closure of categories is reached. This hierarchy may be cyclical. Viewing the aggregation hierarchy of an AdaKNET domain model often gives the best view of how categories are composed of other categories. Therefore, the aggregation hierarchy is more important than the specialization hierarchy in many reuse libraries which describe software subsystems.

Figure 6 shows an example of an aggregation hierarchy for a domain model category named Human. It follows the relationships which describe a human to their types, and then follows the relationships from those types to their types, etc. until there are no more relationships. In this example, the aggregation hierarchy shows that humans have height and weight, and are composed of a head, zero to two arms, zero to two legs, and that arms have hands which have fingers, and legs have feet which have toes. From this example, it can be seen why the aggregation hierarchy can be most important when extracting an entire subsystem from a reuse library.

Multiple Inheritance and Relationships

When a relationship is inherited from multiple categories it is necessary to merge the range and value restrictions of the relationship so that it continues to describe both parent relationships. The relationship's range must be the largest possible range that falls within all the parents' ranges for

the relationship. Similarly, the relationship's type must be the same as or subsumed by all of the parents' types for the relationship. If a range or type meeting these criteria does not exist, the inheritance is not possible without violating subsumption, and the specialization is not allowed. The library model specification translator will report an error when constructing the library model representation and abort.

Any parent relationships which have the same name but do not descend from a common ancestor are distinct relationships. In order for these relationships to be inherited by a single concept, the name conflict must be resolved by renaming one of the relationships before the child concept can be created. This should be done in the LMDL specification for the library model.

Filling Relationships

RLF library models distinguish between two kinds of relationships: "generic relationships" and "particular relationships". Generic relationships are owned by categories, while particular relationships are owned by objects. Particular relationships become "filled relationships" when they represent a particular relationship between two objects. Both the owner and "filler" must be objects.

Filled relationships are depicted just like relationships in the library model graphical notation except that the box in a relationship's circle is filled in. Filled relationships also express a three-way relation between the particular relationship, its owner, and the object "filler" which "fills" the type of the relationship. In figure 5, a filled relationship is shown between **"Heap Ada"**, **written_in**, and **Ada**. **Ada** satisfies the relationship **written_in** for **"Heap Ada"**.

The filler of an object's relationship must adhere to the relationship's restrictions. A filler is said to "satisfy" a particular relationship if the filler is an object that individuates the type of the relationship (either directly or indirectly), and the number of fillers which satisfy the relationship is not already equal to the upper bound of the relationship's cardinality. The number of fillers cannot exceed the upper bound of the relationship's range.

Differentiation

Relationship "differentiation" allows a relationship to be described in a more detailed way than is possible with a single relationship. Differentiation can be thought of as specialization of relationships and is denoted via the "differentiates" relation. Consider the example in figure 7. One of the properties of a **Hand** is that it has **Fingers**. This is modeled by having a relationship **fingers_of** with type **Finger** and owner **Hand**. Relationship differentiation can be used to make finer distinctions; for example, we may want to show that one finger on the hand is a **Thumb** and another is a pinky. Using differentiation, we can do this by creating the subrelationships **thumb_of** and **pinky_of**. The relationships describing these subrelationships may have their own types and ranges to further restrict the kind and cardinality of fillers for the subrelationships. For example, the category **Thumb** appears in the model so the **thumb_of** subrelationship restricts the range of **fingers_of** to **Thumb** for **thumb_of**. No category is modeled for a pinky, however, so the type of **pinky_of** remains **Finger**. Also the ranges of **thumb_of** and **pinky_of** have been narrowed. Thus, differentiation allows one to categorize relationship fillers, and to apply additional restrictions on fillers in those categories.

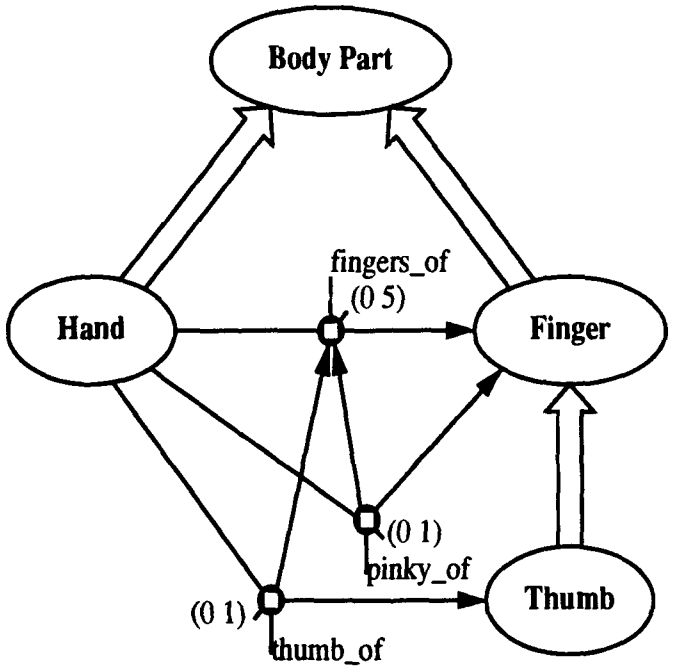

Figure 7: An Example of Relationship Differentiation

RLF library models support two forms of relationship differentiation - "partitioning" and "subsetting". In the first, the immediate differentiators of a relationship "partition" that relationship, i.e., every filler of the differentiated relationship is a filler of exactly one of the subrelationships indicated by the differentiators. In our example, differentiating using partitioning implies that every **Finger** of a **Hand** is the **Thumb** or pinky, and that there are no other fingers that are not a **Thumb** or pinky.

The second form of relationship differentiation, subsetting, is less restrictive than partitioning, allowing one to create subrelationships that do not fully cover all fillers of the differentiated relationship. If our example was created using this subsetting form of differentiation, we could have a filler of **fingers_of** that is not a filler of either **thumb_of** or **pinky_of**. Obviously, the less restrictive subsetting form of relationship differentiation is the one desired in figure 7. Note that an object can be used as a filler of more than one subrelationship (as well as the differentiated relationship itself, in the case of subsetting).

Range checking differs between the two styles of differentiation. In both schemes, the sum of any subrelationship's upper range bound and the other subrelationships' lower range bounds must not exceed the upper range bound of the differentiated relationship. This is because it is impossible to not exceed the differentiated relationship's upper range bound while having the maximum number of fillers for such a subrelationship and adhering to the range restrictions of the other subrelationships. Partitioning also requires that the sum of any subrelationship's lower range bound and the other subrelationships' upper range bounds not be less than the lower range bound of the differentiated relationship. Otherwise it is impossible to cover the differentiated relationship while having the minimum number of fillers for such a subrelationship and adhering to the range restrictions of the other subrelationships. The subset form of differentiation does not impose this last restriction, because the differentiated relationship can have fillers that are not fillers of the differentiating subrelationships.

Finally, if a relationship is differentiated, the entire differentiation is inherited; that is, a subcategory or object of the category where the relationship is differentiated inherits the differentiated relationship, the subrelationships, and the differentiation relation among them. Because of this, it is not possible to differentiate a relationship which has been differentiated with partitioning at a subsuming category. Using differentiation with subsetting, further subsetting of the differentiated relationship is allowed at subsumed categories. In figure 7, this would denote differentiation of the set of those fillers that do not satisfy one of the existing subrelationships, `thumb_of` and `pinky_of`. For example, a `ring_finger_of` subrelationship could be created by the subsetting form of differentiation at a subcategory of `Hand`. With either subsetting or partitioning, subrelationships can be differentiated creating subrelationships of subrelationships.

3.3.5 Attributes

Without the ability to tie the library domain model to the assets it describes, RLF would only provide a domain model encoding and domain understanding tool. To function as a reuse library it is necessary to be able to view and extract the assets modeled in the reuse library model. In RLF, this is done with "attributes."

RLF supports three kinds of attributes: files, strings, and integers. Any number of each of these attributes may be associated with any category or object in the library model. Each attribute has a name which is used to refer to it. Each attribute also has a value whose meaning varies according to the kind of attribute. File attributes' values are filenames. They are expressed relative to the `Text/` directory that is a first-level subdirectory of the directory where the library is built. String and integer attributes' values are strings and integers, respectively, defined with the attribute in the library model specification. Attributes not only serve to connect a library model with the assets it describes, but can also hold data about the library domain which helps describe it, e. g. a string attribute which holds the name of a person or organization.

Attributes are not inherited along the specialization hierarchy like relationships and actions. They are specific to the category or object where they are defined. This is appropriate if you consider an example. Suppose each category in the library model has associated with it a file attribute which contains help information describing that particular category. These attributes should not be available at subcategories or objects of the category where they are defined because they do not describe the category or object where they were inherited.

Attributes are also essential for the operation of the RLF action mechanism (see 3.3.6). The string attributes of action categories are used when invoking the action. These attributes either represent strings to be processed and executed in a UNIX *csh* shell or special strings used to identify built-in procedure actions. Action targets which provide values or files for actions to operate on, and action agents which allow action invocation to be tailored at any category or object, are also implemented using attributes.

3.3.6 Actions

The AdaKNET representational foundation of RLF has been enhanced beyond the representational capabilities of a standard structured inheritance network to include the ability of associating actions with categories or objects. This allows library models to interface with application-specific code or

perform operating system functions based on invocation of actions by the `Graphical_Browser` or `Library_Manager` applications. In the context of the RLF, this allows actions to be performed on reuse library assets which have been modeled in the underlying library domain model. For more information on action invocation than is found here, consult the **RLF User's Manual** and the **RLF Administrator's Manual.**

An "action" can be associated with a category or object and has five parts. The first part is simply a tag which serves as the name of the action used to discriminate it from other actions. The second part is the "action category" which is the name of a category which exists in the library model's "action sub-model." (The action sub-model is detailed in the next subsection.) The action category is used to describe the specifics of the action and any relation it may have to other objects in the network. In the most common use of actions, the action category has a string attribute associated with it which is a string that is executed in an operating system shell when the action is invoked. The third part of an action is the optional one or more "action targets", or "targets", which are strings which represent the context the action will act upon. The action targets are the names of attributes associated with the category where the action appears. The fourth part is the optional one or more "action agents", or "agents", which are strings which represent modifiers to the action invocation. The action agents, like targets, are the names of attributes associated with the category or object where the action appears. The final part of an action is a "privileged" flag, which can be used by the RLF application to prune the selection of available actions based on the user's orientation. This flag distinguishes actions that can be invoked by a reuse library administrator but not by a reuse library user.

Actions are inherited by categories in much the same manner as relationships. All subcategories which are subsumed by a category which declares an action will have that action available. Actions can also be inherited along several specialization links in the case of multiple inheritance.

The Action Sub-Model

For the RLF action invocation mechanism to operate correctly, each library model must contain a sub-model which describes the action categories of all the actions in the library model. The action sub-model must have a certain structure. It is rooted at a category named `"Action Definition"`. `"Action Definition"` should have at least a child named `Action` and a child named `"Action Type"`. The category `Action` must have at least a relationship named `has_action_type` with type `"Action Type"`. The category `"Action Type"` must have at least a child category named `"System String"` and should usually have another child named `"Ada Procedure"`. All action categories referenced from action definitions in the main library model must be a child of the category `Action` and must restrict the type of the relationship `has_action_type` to be a child of `"Action Type"`, usually `"System String"`. An example action sub-model can be found in appendix C in the starter library model template. Subsets of this library model fragment can be found in any of the example libraries included with RLF. The action sub-model of the sort and search algorithms library model used for most examples in this manual can be found in section 4.1.6. A list of all reserved entity names in RLF appears in table 1.

Action Restriction

Actions, again much like relationships, can also be restricted at subcategories below the category where they are declared. The action category part of an action can be made more specific as long as the new action category is a subcategory of the inherited action category. In this way, an action

Identifier	Defined Purpose
"Action Definition"	Root category of a model's action sub-model.
Action	Immediate child of "Action Definition". Parent of all model action categories.
"Action Type"	Immediate child of "Action Definition". Parent of all model action types.
"System String"	Immediate child of "Action Type". Pre-defined library model action type.
"Ada Procedure"	Immediate child of "Action Type". Pre-defined library model action type.
has_action_type	Relationship at an action category whose type is a child of "Action Type".

Table 1: Reserved Entity Names in RLF

can be made more specific as the category at which it exists becomes more specialized. A non-privileged inherited action can also be made privileged, but following the requirement that nothing in the structured inheritance network should ever be made more general lower in the specialization hierarchy, a privileged action cannot be made non-privileged at a subcategory inheriting the action. Action targets can be renamed at subcategories, regardless of inheritance. These are strings particular to a category which do not rely on inherited information. If an inherited action at a category or object does not explicitly name its action targets then the explicit action targets of its closest ancestor with targets will be inherited. If the action explicitly names even one target then any inherited targets are overridden. Action agents are handled exactly like action targets when it comes to inheritance and restriction.

Action Invocation

Invoking an action at a category or object in the library model from an RLF application follows certain semantics. Once the action has been selected by name (or has defaulted when there is only one choice), the action's action category is examined. The action category is described with a relationship named has_action_type which indicates what type of action it is. The has_action_type relationship is inherited from the reserved category Action which must be an ancestor of any action category, either directly or indirectly. The action category must restrict the has_action_type relationship to indicate one of the two types of actions in RLF. One type calls an internal Ada procedure linked into the RLF application. The second type, and most common, manipulates a string attribute found at the action's action category and then executes the resulting string inside a UNIX *csh* shell. The has_action_type relationship indicates which type of action invocation to do for a particular action. More information on modeling actions can be found in section 8.2. More information on attributes can be found in section 3.3.5.

Action Invocation - System Strings

When invoking an operating system string type of action, RLF goes through several processes. It will obtain the string attribute from the action's action category and process it in several ways and then execute it in a UNIX *csh* shell. It will process the string for each action target, and for each target, further manipulate the string according to any action agents present.

When processing the action targets of an action, RLF collects the set of targets defined with the action which have attributes of the same name at the category or object where the action is being invoked. If no explicit action targets are defined at that category or object, then the action invocation mechanism will use any action targets that have been inherited from subsuming

categories. If the action is newly defined at the category or object where it is being invoked and has no targets, or no action targets are inherited from supercategories, then the action is considered "targetless", and the string attribute found at the action's action category will not be modified to include a target file or value to operate on. If one action target exists, then the action invocation mechanism will take the target name and search for an attribute at the category or object with the same name. It will then substitute the string value or filename represented by the attribute for a special marker in the string attribute of the action category. If there exists a list of action targets, RLF will repeat the entire action invocation process once for each target in the list of targets.

After processing the action category's string attribute to insert the action target, if necessary, the action invocation mechanism will then process any action agents. Any action may have up to nine action agents which are represented in the action category's string attribute with numbered markers. In a manner very much like the substitution of the action target into the string to be executed, agent values are collected from string or file attributes at the category or object where the action is invoked. The attributes substituted for the numbered markers in the action category's string attribute are the ones having the same names as the agents in the action's agent list. In there is no agent list for the action, no agent substitution is done. If there are one or more agents, they are substituted into the string to be executed for each action target processed. The order that any action agents appear in the action's agent list is the number of the action agent marker for which the agent is substituted. After the string has been processed for a target and all agents, it is executed in a UNIX *csh* shell. The string is executed once for each target in the target list, or only once if the action has no targets.

Action Invocation - Internal Ada Procedures

When invoking the internal procedure call type of action, RLF will first try to match the string attribute of the action's action category against a list of built-in action procedures. If the string does not match one of those in the list, the action invocation fails. If the string matches one in the list, then the internal procedure is executed with any **stderr** or **stdin** results written to the UNIX shell from which the RLF application was first executed.

The RLF currently supports four internal Ada procedure actions, **Extract**, **Import**, **Export**, and **"Display Attributes"**. The Ada procedure implementing **Extract** will copy all the file attributes at the category or object where it is invoked to the directory specified by the environment variable **RLF_WORKING_DIR**. If this variable is not set it will copy the files to the current working directory. The ability of the procedure to copy the files described by the file attributes will depend on the correct file permissions having been set in the operating system. The procedure implementing the **"Display Attributes"** action will invoke a pager program to view all the attributes associated with the category or object where the action was invoked. The program starts up viewing a table of contents of all the attributes at the category or object. String and integer attribute values are displayed directly in this table of contents. The files described by file attributes are viewed in their entirety as separate files within the pager program. Instructions for changing the program to view the separate file contents are given on the table of contents page. The Ada procedures implementing the **Import** and **Export** actions are not yet fully implemented.

How these built-ins are explicitly referenced when modeling an RLF reuse library is discussed in section 8.2.1.

4 The Library Model Definition Language (LMDL)

This section introduces and describes the Library Model Definition Language (LMDL). LMDL is the mechanism of choice for instantiating AdaKNET knowledge models. Although it is possible to simply write a program which instantiates AdaKNET via a sequence of calls to the AdaKNET ADT package, this procedure is very difficult and not descriptive of the knowledge model being constructed. Because of the strict semantics that ensure the integrity of RLF library domain models, construction of the AdaKNET representation of the model must be done in a specific order to make sure AdaKNET entities are present when they are referenced. With LMDL, the need to define AdaKNET entities before they are referenced has been eliminated. For example, a category can appear as the type for a relationship before the category is defined in the specification. In previous experience with construction of semantic network models by interactive editors, the required order of creation has proved somewhat non-intuitive (e.g., the specialization hierarchy must be built top-down, but relationships and fillers must be added bottom-up). Relaxing the requirement of definition before use in the LMDL language should provide several advantages. Library model definitions can be organized in the most easily comprehensible way for the modeler. Specifications can be modularized more easily, while preserving the overall integrity checking on the model. For example, a category definition can include contiguous definition of the local and restricted relationships of the concept; this would not be possible with a definition-before-use scheme. LMDL facilitates description of a library domain model and hopefully makes the specification easily maintainable.

Besides providing a convenient and readily modifiable medium for defining models, LMDL also promotes the reuse of knowledge models as assets in their own right. Since LMDL descriptions are ASCII text, AdaKNET domain models described via LMDL specifications can be transported "as is" to any site. Also, as will be seen, the language design of LMDL contains features for modularization of knowledge models; this will be important in the reuse of knowledge models via amalgamation of small, special-purpose sub-models. For example, a fragment of a domain model describing Ada data types could be usefully integrated in other library models of tools sensitive to Ada type semantics.

The chosen form of the LMDL language provided some interesting implementation challenges. It requires a fairly complex translator implementation, since a single-pass translation will not be able to do adequate consistency checking. Here, our use of SSAGS, a Paramax-proprietary meta-generation system based on ordered attribute grammars [PKP+82], is a key element to the feasibility of our approach. LMDL will evolve as experience shows what organizing schemes are most appropriate for specifying domain models. A major advantage of our approach is that we are not committing to a particular organizational approach, but rather implementing a flexible specification language. Modelers will be able to use this language, not only to rapidly prototype reuse library models, but to explore different definitional strategies as well. This methodological work is an essential prerequisite to the use of library domain model specifications as reusable assets.

4.1 Representation of AdaKNET Entities

This section presents the language syntax for each of the library model entities described in section 3. It also shows how library advice modules, or "inferencers," are attached to the library model in a LMDL specification. See section 5 for more on inferencers. Syntax is presented in EBNF (Extended Backus-Naur Form). This is the same notation used throughout the Ada Language Reference Manual (LRM) [Ada83]. A brief description follows. A more complete description appears in section 1.5 of the Ada LRM.

lower_case_word
: nonterminal (e.g. library_model_spec).

*italicized_part*_lower_case_word
: refers to same nonterminal as the lower case word without italicized part. The italicized part is used to convey some semantic information (e.g. *category*_name).

bold_face_word
: language token (e.g. **category**).

`{item}`
: braces enclose item which may be repeated zero or more times.

`[item]`
: brackets enclose optional item.

`item1 | item2`
: alternation; either item1 or item2

A language syntax for LMDL without additional annotations is located in appendix A of this manual. A table of reserved identifers can be found in section 3.3.6 in table 1.

4.1.1 Library Models

RLF supports the simultaneous existence of many individual libraries and library domain models. Thus, LMDL provides a single language construct, the library model, for encapsulating the description of a library domain model. The general form of library model definitions is:

library_model_spec ::=
library model *model*_name **is**
[incremental_indication]
[root_category]
{category_or_object_or_inferencer}
end model_name ;

The LMDL library model construct roughly parallels the Ada package construct. Both describe a named unit which encapsulates related information.

An example of a library model definition from the sort and search algorithms library domain model specification:

```
library model "Sort and Search Algorithms" is

   -- category and object definitions
   -- and inferencer attachments

end "Sort and Search Algorithms";
```

LMDL also supports the addition of additional AdaKNET entities to an existing RLF reuse library through the use of a partial model specification. This is sometimes referred to as "incremental" LMDL. Incremental LMDL specifications differ from normal LMDL specifications by incorporating the following syntax:

```
incremental_indication =
        'extend' 'library' 'model' model__name ';'
```

In addition to this syntax, which specifies the name of the pre-existing library which will be extended, incremental LMDL specifications can not make root category definitions other than to add new definitions to the root category definition of the library being extended.

An example of an incremental LMDL specification which adds math function algorithms to the sort and search algorithms library might look like this:

```
library model "Sort, Search, and Math Algorithms" is

   extend library model "Sort and Search Algorithms";

   -- additional category and object definitions
   -- and inferencers attachments describing math functions
   -- connected to categories and objects in the
   -- sort and search algorithms library model

end "Sort, Search, and Math Algorithms";
```

Translating this specification would create a new library named "Sort, Search, and Math Algorithms" containing all the constructs of the sort and search algorithms library plus any additional entities describing math function algorithms.

4.1.2 Categories

Categories are the basic descriptive construct in LMDL. They represent classes of things in the domain model. The syntax for defining categories in LMDL is as follows:

```
category ::=
      category category_name ( specializes ) is
        [local_relationships]
        [restricted_relationships]
        [differentiated_relationships]
        [local_attributes]
        [local_actions]
      end category ;

specializes ::=
    category_name {, category_name}
```

The category syntax indicates the position of a category in the library model specialization hierarchy via its supercategories and provides a mechanism for encapsulating any local relationships, attributes, or actions and any constraints on inherited relationships or actions at the category.

A category may be distinguished from its supercategories by restrictions on inherited relationships and by new relationships introduced at the category. New actions or inherited actions constrained with new privilege or targets or a more specialized action category also help differentiate categories. Attributes defined at a category also distinguish it, especially since attributes are always local and are not inherited. A bracketing syntax is used for these sub-structures in LMDL to set them apart from other defining constructs:

- Aggregation via **relationships ... end relationships.**
- Restriction via **restricted relationships ... end restricted.**
- Differentiation via **differentiated relationships ... end differentiated.**
- Attribute association via **attributes ... end attributes.**
- Action association via **actions ... end actions.**

One special LMDL syntax is reserved for the root category of the library domain model. This is the most general class of things in the library model from which all other categories are descended in the library's specialization hierarchy. The syntax for the root category in LMDL is as follows:

root_category ::=
root category *category_name* **is**
[local_relationships]
[differentiated_relationships]
[local_attributes]
[local_actions]
end root category ;

Restricted relationships are not allowed at the root category since any relationship there is necessarily local and not inherited. It therefore should be initially defined with the intended range and type restrictions.

An example of some category definitions from the sort and search algorithms library domain model specification:

```
root category Thing is
end root category;

--

category Algorithms ( Thing ) is
   relationships
      is_written_in (0 .. infinity) of "Source Language";
      works_on (0 .. infinity) of "Data Structure";
      has_best_case_of (0 .. 1) of Performance;
      has_avg_case_of (0 .. 1) of Performance;
      has_worst_case_of (0 .. 1) of Performance;
      has_size_of (0 .. 1) of "Lines of Code";
   end relationships;
end category;

--

category "Data Structure" ( "Attribute Values" ) is
end category;

--

category View ( Action ) is
   restricted relationships
      has_action_type of "System String";
   end restricted;
   attributes
      string is "xterm -e $RLF_PAGER ## &";
   end attributes;
end category;
```

4.1.3 Objects

Objects describe members of the classes described by categories in LMDL. The syntax for defining objects in LMDL is as follows:

object ::=
object *object*_name (individuates) **is**
[restricted_relationships]
[differentiated_relationships]
[satisfied_relationships]
[local_attributes]
[local_actions]
end object ;

individuates ::=
*category*_name {, *category*_name}

The object syntax indicates the position of an object in the library model hierarchy via the list of supercategories it individuates. An object can further describe the member of the parent category it represents with additional constraints on inherited relationships and actions, and with new attribute definitions.

A bracketing syntax is used for this additional sub-structure in LMDL to describe the which other objects fill particular relationships for the object being defined:

- Satisfaction via **fillers** ... **end fillers**.

An example of some object definitions from the sort and search algorithms library domain model specification:

```
object "Example Shaker Sort" ( Shakersort ) is
   restricted relationships
      is_written_in (1 .. 1) of "Source Language";
      works_on (1 .. 1) of "Data Structure";
      has_avg_case_of (1 .. 1) of Quadratic;
      has_worst_case_of (1 .. 1) of Quadratic;
      has_size_of (0 .. 1) of Number;
   end restricted;
   fillers
      Ada satisfies is_written_in;
      Array satisfies works_on;
      Eleven satisfies has_size_of;
   end fillers;
   attributes
      file desc_source is "sort_and_search/exchange_sort_desc";
      file source is "sort_and_search/shaker_sort_.a";
```

```
      string size_of is "11";
   end attributes;
   actions
      "View Source" is View on source;
      "View Code Size" is "Display Integer" on size_of;
   end actions;
end object;

--

object "Example Quicksort" ( Quicksort ) is
   restricted relationships
      is_written_in (1 .. 1) of "Source Language";
      works_on (1 .. 1) of "Data Structure";
      has_worst_case_of (1 .. 1) of Quadratic;
      has_size_of (0 .. 1) of Number;
   end restricted;
   fillers
      Ada satisfies is_written_in;
      Array satisfies works_on;
      "N^2" satisfies has_worst_case_of;
      "Twenty-Four" satisfies has_size_of;
   end fillers;
   attributes
      file desc_source is "sort_and_search/exchange_sort_desc";
      file source is "sort_and_search/quick_sort_.a";
      string size_of is "24";
   end attributes;
   actions
      "View Code Size" is "Display Integer" on size_of;
      "View Source" is View on source;
      "Extract Source" is Extract on source;
   end actions;
end object;
```

4.1.4 Relationships

Relationships are the primary construct in LMDL used to differentiate categories from one another and describe the class of things in the library domain model which the categories model. They also provide more specific information which describes how objects fit exactly into the category they individuate and which other objects help describe the objects.

Relationship definition, restriction, differentiation, and filling (or satisfaction) occurs in bracketed blocks within category and object definitions in LMDL. The syntax for these relationship definitions in LMDL is as follows:

local_relationships ::=
relationships
relationship {relationship}
end relationships ;

relationship ::=
*relationship*_name (number .. number_or_infinity)
of *category*_name ;

restricted_relationships ::=
restricted relationships
restriction {restriction}
end restricted ;

restriction ::=
range_restriction | value_restriction |
range_and_value_restriction

range_restriction ::=
*relationship*_name (number .. number_or_infinity) ;

value_restriction ::=
*relationship*_name **of** *category*_name ;

range_and_value_restriction ::=
*relationship*_name (number .. number_or_infinity)
of *category*_name ;

differentiated_relationships ::=
differentiated relationships
differentiation {differentiation}
end differentiated ;

differentiation ::= subset | partition

subset ::=
subset *relationship*_name **into**
relationship {relationship}
end subset ;

partition ::=
partition *relationship*_name **into**
relationship {relationship}
end partition ;

satisfied_relationships ::=
fillers
filler {filler}
end fillers ;

filler ::=
*object*_name **satisfies** *relationship*_name ;

This syntax first addresses the initial definition of a relationship, and then the specifics of how relationships are restricted in range and/or type, how they are differentiated by partitioning or subsetting, and finally how relationships are filled (or satisfied) by other objects in the library model that fit the relationship type and range.

Some examples of LMDL syntax (not all from the same library model specification) showing relationship definitions follow:

```
category Algorithms ( Thing ) is
   relationships
      is_written_in (0 .. infinity) of "Source Language";
      works_on (0 .. infinity) of "Data Structure";
      has_best_case_of (0 .. 1) of Performance;
      has_avg_case_of (0 .. 1) of Performance;
      has_worst_case_of (0 .. 1) of Performance;
      has_size_of (0 .. 1) of "Lines of Code";
   end relationships;
end category;

--

object "Heap Ada" ( Heapsort ) is
   restricted relationships
      is_written_in (1 .. 1) of "Source Language";
      works_on (1 .. 1) of "Data Structure";
      has_worst_case_of (1 .. 1) of Linearithmic;
      has_avg_case_of (1 .. 1) of Linearithmic;
      has_size_of (1 .. 1) of Number;
   end restricted;
   fillers
      Ada satisfies is_written_in;
      Array satisfies works_on;
      "N * log (N)" satisfies has_worst_case_of;
      "(N / 2) * LOG (N)" satisfies has_avg_case_of;
      Eighteen satisfies has_size_of;
   end fillers;
   attributes
      file desc_source is "sort_and_search/selection_sort_desc";
```

```
      file source is "sort_and_search/heap_spec_.a";
      string size_of is "18";
   end attributes;
   actions
      "View Code Size" is "Display Integer" on size_of;
      "View Source" is View on source;
   end actions;
end object;

--

category Hand ("Body Part") is
   relationships
      fingers_of (0 .. 5) of Finger;
   end relationships;
   differentiated relationships
      subset fingers_of into
         pinky_of (0 .. 1) of Finger;
         thumb_of (0 .. 1) of Thumb;
      end subset;
   end differentiated;
end category;
```

4.1.5 Attributes

The attribute construct of LMDL is used to define the string, file, and integer attributes associated with a category or object in the library domain model. Attributes are particular to the category or object where they are defined and are not inherited. String attributes at categories below the reserved category `Action` also have special meaning when actions referencing those action categories are invoked (see 3.3.6).

The LMDL syntax for defining category and object attributes is as follows:

local_attributes ::=
attributes
attribute {attribute}
end attributes ;

attribute ::=
string_attribute |
file_attribute |
integer_attribute

string_attribute ::=
string [name] **is** string_literal ;

file_attribute ::=
file [name] **is** filename ;

integer_attribute ::=
integer [name] **is** number ;

String and integer attributes are internalized when the library model specification is translated. File attributes represent filenames relative to the **Text/** directory which is a first-level subdirectory below the directory where the library is being constructed.

Some examples of attribute definitions from the sort and search algorithms library model specification follow.

```
category View ( Action ) is
   restricted relationships
      has_action_type of "System String";
   end restricted;
   attributes
      string is "xterm -e $RLF_PAGER ## &";
   end attributes;
end category;

--

category "Insertion Sorts" ( "Internal Sorts" ) is
   attributes
      file desc_source is "sort_and_search/insertion_sort_desc";
   end attributes;
   actions
      "Read Description" is "Display Description" on desc_source;
   end actions;
end category;

--

object "Heap Ada" ( Heapsort ) is
   restricted relationships
      is_written_in (1 .. 1) of "Source Language";
      works_on (1 .. 1) of "Data Structure";
      has_worst_case_of (1 .. 1) of Linearithmic;
      has_avg_case_of (1 .. 1) of Linearithmic;
      has_size_of (1 .. 1) of Number;
   end restricted;
   fillers
      Ada satisfies is_written_in;
      Array satisfies works_on;
      "N * log (N)" satisfies has_worst_case_of;
```

```
      "(N / 2) * LOG (N)" satisfies has_avg_case_of;
      Eighteen satisfies has_size_of;
   end fillers;
   attributes
      file desc_source is "sort_and_search/selection_sort_desc";
      file source is "sort_and_search/heap_spec_.a";
      string size_of is "18";
   end attributes;
   actions
      "View Code Size" is "Display Integer" on size_of;
      "View Source" is View on source;
   end actions;
end object;
```

4.1.6 Actions

Actions are the LMDL construct used to define actions that can be performed on attributes of categories or objects in the library model. They are inherited like relationships and can be constrained when inherited. The action's action category can be made more specific, non-privileged actions can be made privileged, and inherited action targets and agents can be added, or overridden and explicitly named locally.

The syntax for actions in LMDL is as follows:

local_actions ::=
actions
action {action}
end actions ;

action ::=
*action*_name **is** [**privileged**] *category*_name
[**on** targets] [**with** agents] ;

targets ::=
*target*_name {, *target*_name}

agents ::=
*agent*_name {, *agent*_name}

The interpretation of the parts of an action when the action is invoked is covered in detail in section 3.3.6. For information on modeling actions and action invocation with the PCTE version of RLF, consult appendix E.

An example of action definitions from the sort and search algorithms library model specification follow.

```
category "Insertion Sorts" ( "Internal Sorts" ) is
   attributes
      file desc_source is "sort_and_search/insertion_sort_desc";
   end attributes;
   actions
      "Read Description" is "Display Description" on desc_source;
   end actions;
end category;

--

object "Heap Ada" ( Heapsort ) is
   restricted relationships
      is_written_in (1 .. 1) of "Source Language";
      works_on (1 .. 1) of "Data Structure";
      has_worst_case_of (1 .. 1) of Linearithmic;
      has_avg_case_of (1 .. 1) of Linearithmic;
      has_size_of (1 .. 1) of Number;
   end restricted;
   fillers
      Ada satisfies is_written_in;
      Array satisfies works_on;
      "N * log (N)" satisfies has_worst_case_of;
      "(N / 2) * LOG (N)" satisfies has_avg_case_of;
      Eighteen satisfies has_size_of;
   end fillers;
   attributes
      file desc_source is "sort_and_search/selection_sort_desc";
      file source is "sort_and_search/heap_spec_.a";
      string size_of is "18";
   end attributes;
   actions
      "View Code Size" is "Display Integer" on size_of;
      "View Source" is View on source;
   end actions;
end object;
```

Actions are strictly defined by action categories which exist in a section of the library model specification called the "action sub-model." The action sub-model must contain certain categories and relationships for the RLF action invocation mechanism to work. A table of reserved identifers from the action sub-model can be found in section 3.3.6 in table 1. The action sub-model from the sort and search algorithms library model specification, including the action categories of the actions defined in the example above, follows:

```
category "Action Definition" ( Thing ) is
end category;
```

```
category "Action Type" ( "Action Definition" ) is
end category;

   category "System String" ( "Action Type" ) is
   end category;

   category "Ada Procedure" ("Action Type") is
   end category;

category Action ( "Action Definition" ) is
   relationships
      has_action_type (1 .. 1) of "Action Type";
   end relationships;
end category;

   category View ( Action ) is
      restricted relationships
         has_action_type of "System String";
      end restricted;
      attributes
         string is "xterm -e $RLF_PAGER ## &";
      end attributes;
   end category;

      category "Display Description" ( View ) is
         restricted relationships
            has_action_type of "System String";
         end restricted;
         attributes
            string is "xterm -e $RLF_PAGER ## &";
         end attributes;
      end category;

      category "Display Integer" ( View ) is
         restricted relationships
            has_action_type of "System String";
         end restricted;
         attributes
            string is "xterm -e $RLF_PAGER ## &";
         end attributes;
      end category;

   category Extract ( Action ) is
      restricted relationships
         has_action_type of "Ada Procedure";
      end restricted;
      attributes
         string is "Extract Asset";
```

```
        end attributes;
    end category;
```

This example also shows the use of "substitution markers" in the string attributes of action category definitions. When a "**System String**" type action is invoked, substitution markers in the action category's string attribute are substituted with an action target and action agents from the action definition at the category or object where the action is invoked. The marker "##" is used as a placeholder for the action target. It is replaced with a string value or filename on which the action will operate. Markers of the form "%%*n*", where *N* is a numeral from 1 to 9, are placeholders for the action's action agents. They are replaced with string values or filenames which modify the action string for each target. The order of the action agent in the action definition's agent list is used to determine which "%%*n*" substitution marker is replaced. More information on how substitution markers are replaced when an action is invoked is found in section 3.3.6.

The following example shows an action category definition in LMDL which uses a mix of target and agent substitution markers to define a general print action in a library model.

```
category Print ( Action ) is
-- this action category describes a general print action
   restricted relationships
      has_action_type (1 .. 1) of "System String";
   end restricted;
   attributes
      -- ## marks the file to be printed
      -- %%1 marks the UNIX print command to use
      -- %%2 marks any options to the print command
      -- also run the action in the UNIX background
      -- so the RLF application continues
      string print_command is "%%1 %%2 ## &";
   end attributes;
end category;
```

4.2 Attaching Inferencers in LMDL

RLF has the ability to attach library advice modules, or "inferencers", to categories or objects in the library domain model. Inferencers are described in section 5. This association between inferencers and categories or objects is specified in LMDL where all category and object definitions are available and the modeler can easily see the associations between the library model and its advice modules.

Inferencer attachments in LMDL identify the name of the inferencer and the name of the category or object it is associated with. The LMDL syntax for attaching inferencers is as follows:

```
inferencer ::=
      attach inferencer inferencer_name
        to category_or_object_name ;
```

*category_or_object*_name ::=
*category*_name | *object*_name

Inferencer attachments are usually located in the library model specification immediately following the definition of the category or object to which they are attached. Also, to diminish confusion and mismatching of names, the inferencer is usually named similarly to the category or object it is associated with.

NOTE: The names of inferencers expressed in inferencer attachment LMDL clauses must appear in all lower case. This is because the RBDL translator currently converts all identifiers in the inferencer's RBDL specification to lower case.

Some examples of inferencer attachment to library categories or objects follow. The examples are taken from the sort and search algorithms library model specification.

```
category "Straight Selection" ( "Selection Sorts" ) is
   attributes
      file desc_source is "sort_and_search/selection_sort_desc";
   end attributes;
end category;

attach inferencer straight_selection to "Straight Selection";

category "Exchange Sorts" ( "Internal Sorts" ) is
   attributes
      file desc_source is "sort_and_search/exchange_sort_desc";
   end attributes;
   actions
      "Read Description" is "Display Description" on desc_source;
   end actions;
end category;

attach inferencer exchange_sorts to "Exchange Sorts";
```

4.3 Other Syntax

The syntax for various low-level, LMDL primitives follows. These primitives may be mentioned in the syntax examples above and are included for completeness. A full syntax summary for LMDL is given in appendix A.

name ::= identifier | string_literal

filename ::= string_literal

identifier ::= letter {[underline] letter_or_digit}

letter_or_digit ::= letter | digit

letter ::= upper_case_letter | lower_case_letter

number_or_infinity ::= number | **infinity**

number ::= digit {digit}

5 AdaTAU Library Model Advice

5.1 Overview

AdaTAU is a rule-based inferencing subsystem written in Ada based on TAU, a Paramax-proprietary, rule-based production system that incorporates an agenda mechanism for directing interaction with a user along with a forward-chaining inference system. TAU is an acronym made up of the first letters of the phrase that summarizes the organization of this component — *Think, Ask, Update*. *Think* refers to the analysis of the fact base which is used to record information about the domain under consideration, and to the application of rules which directly modify this fact base (add or delete facts for example). Rules considered during this phase may also lead to the scheduling of queries which will be processed subsequently. *Ask* denotes the capability of posing questions, and recording responses in answer to the questions, that are scheduled as a result of the *Think* phase. Finally, the *Update* phase will modify the fact base in a manner that depends on the responses recorded in the *Ask* phase. The concepts and terms used in this overview are presented in the next sections.

5.1.1 Rule-based Inferencing

"Inferencing" is taking certain known information, called "facts" here, and inferring new facts. "Rule-based inferencing" is a type of inferencing structured with the use of "rules." Rules are control structures which state the association between true and possibly true information. For example, a simple "*if A, then B* " structure is a rule which states if *A* is true then *B* is true. The rules which AdaTAU uses in its rule-based inferencing are discussed in detail in section 5.2. The collection of all the facts and rules in an instance of AdaTAU is called an "inference base."

Rule-based inferencing subsystems can also be thought of as providing an "inference engine." This term is used since the interpretation of rules is cyclic, with information made true by previous rules being used to trigger other rules, over and over until no more rules can be triggered. AdaTAU "drives" this iteration of rules and facts and thus is called an inferencing "engine."

AdaTAU's inference engine is "forward-chaining," meaning rules structure the inference from front to back in an "*if A, then B* " fashion. The facts which make up *A* are called the "antecedents," and the facts which make up *B* are called the "consequents." Some inferencing engines are "backward-chaining," meaning they structure the inference from the consequents to the antecedents. This is sometimes useful when the user has certain goal facts. The rules are traversed backwards until facts are reached which are known to be true, proving the goal facts true. Some inferencing engines are hybrids of forward- and backward-chaining systems. Currently, AdaTAU only provides forward-chaining rule-based inferencing.

Monotonic/Non-Monotonic Reasoning

In the previous discussion, inferencing with rules was limited to the addition of new facts. Such an inference system is called a "monotonic" inference system. With this system, the number of facts can only grow larger.

In many applications that must model real world representation and manipulation of information, there is a need to handle the deletion (and modification) of facts as well as their addition. Inference systems which support the retraction of information are called "non-monotonic" systems. In non-monotonic systems, the antecedents of a rule may include the absence of facts as a precondition, or have facts in the consequents which should be deleted when the rule is applied. The AdaTAU inference engine is non-monotonic but does not allow the stated absence of a fact to be an antecedent.

When facts are deleted, the effect of deletions can ripple through the fact base with the effect that the conclusions of rules that were used because of the presence of certain antecedent facts can now be considered to be invalid. The facts added to the fact base as a result of these now invalidated rules should themselves be withdrawn. This process continues with several passes through the fact base necessary to bring the fact base into a consistent state.

"Truth maintenance" is that part of the inferencing system that manages the consistency of the information among all the system's facts. An elementary example of truth maintenance is a check that facts which can only represent two states, such as an on/off switch, only have one of those states represented in the collection of all the facts at a time. For example, both of the facts <power, on> and <power, off> cannot be simultaneously part of the collection of facts known to be true. Strictly monotonic inferencing systems essentially require no truth maintenance component, and depending on the nature and generality of the facts, non-monotonic inferencing systems can require very complicated truth maintenance subsystems.

5.1.2 Partitioned Inference Bases

In complex domains, it is probable that there will be a great number of facts and rules in the inference base. The sheer quantity of information would make it difficult for modelers to develop and later maintain a logically consistent inference base. Following the rule and fact dependencies and flow would be very difficult with too many facts and rules.

AdaTAU rule-based inferencing addresses this problem by supporting the concept of "partitioned" inference bases. In this scheme, rules and facts which have logical ties in the domain and be grouped together and kept separate from other rules and facts which are not directly related. This keeps the number of rules and facts in an AdaTAU inference base manageable and topical. AdaTAU supports fact interfaces between inference bases to allow inferencing control to pass between partitioned inference bases when the facts in that inference base indicate that control should shift.

In the context of an RLF reuse library this means that an AdaTAU inference base can be associated with individual categories or objects in the library model. When control passes between partitioned inference bases in AdaTAU, this can be detected and used to cause a shift of the current category in the library model. In this way, the progression of AdaTAU execution of partitioned inference bases implementing library domain advice can be used to navigate the library user toward reusable

assets in the library. The answer to questions posed by the inference bases can direct the library user closer to the goal asset.

5.1.3 Expanded Use of AdaTAU in Libraries

Besides it's use as an advice-giving mechanism for reuse library users, it may be possible for AdaTAU to play a larger role. Some other possible uses for AdaTAU are to keep fact bases containing a user's profile information so that the presentation of the library could be tailored to particular users. Another use might be to collect metric information in fact bases that could later be scanned by the library administrator. AdaTAU inferencers could be used in complex extract operations where configuration of multiple, dependent assets are being extracted to put together an entire subsystem. AdaTAU forms a flexible logic component that can be individually developed for many purposes.

5.2 AdaTAU Concepts

This section contains descriptions of the basic concepts supporting rule-based inferencing in AdaTAU which is used in RLF reuse libraries to provide advice to the user on locating the correct reusable asset. It describes the basic objects of AdaTAU, the more complex constructions for rules, and particulars about the inference engine that AdaTAU provides.

5.2.1 Facts

A "fact" in AdaTAU is any dynamic quantum of information that the system must be able to process. Typically, facts are stored in a fairly rigid form that is designed to provide efficient access for the system. Some common organizational schemes are property-boolean state pairs, or attribute-value pairs, or more generally, triples denoting object names, attribute names, and corresponding values. For example, a fact indicating the current state of a printer's indicator light could be expressed as <printer_indicator_light_on, true>, or <printer_indicator_light, on> or <printer, indicator_light, on>. Fact structures can be considerably more complicated. At one extreme, one can imagine English-like clauses, or arbitrary lists that can themselves contain sublists. For example, <father_of, Peter, <husband_of, Nancy>> can be used to represent the fact that Peter's father is Nancy's husband.

AdaTAU provides an attribute-value structure for facts. Facts are thus viewed as pairs of properties and values of such properties. An attribute can be understood to be the name of a property of the domain under consideration. A value for an attribute provides a characterization of that property. Both attributes and values are implemented simply as strings, although AdaTAU provides a significant management component so that instances of these objects can be restricted and checked for conformance to declared rules for attribute-value pairs. Some examples of facts from some of the inferencers of the sort and search algorithms example library are <ordering, nearly_sorted>, <answer, sorting_algs>, and <continue_confirmed, yes>.

Type	Value Semantics
one_of	exactly one of a specified set of values
some_of	one or more of a specified set of values
reference	exactly one string value representing a reference to the actual value
any	exactly one arbitrary string value

Table 2: AdaTAU Fact Types

5.2.2 Fact Bases

Collections of facts are called "fact bases." In their simplest form, fact bases are sets of attribute-value pairs. The fact base is the repository of all facts that are currently known to be true in an inference base. Further deductions are made from facts in the fact base, and the results of these deductions add or delete facts in the fact base. Inference bases can start the inferencing cycle with some facts already in the local fact base. This is called the "initial fact base." When inferencing control switches between two partitioned inference bases, facts can be "exported" from the local fact base and "imported" into the fact base of the new inference context.

Which facts can be present in a fact base in AdaTAU depends on what facts are currently in the fact base. Facts expressing contradictory information are not allowed. The different types of facts supported by AdaTAU are discussed in the next section. A fact's type dictates how it interacts with other facts in a fact base.

5.2.3 Fact Base Schemas

Each inference base has a "fact base schema." A fact base schema restricts the form and value sets for facts within a particular fact base. The definition for all the facts that can exist in an inference base are given in the fact base schema. A fact's definition includes the fact's name (which forms the attribute part of the attribute-value pair), its type, and its possible values. Table 2 shows the fact types supported by AdaTAU and the semantics associated with each type.

5.2.4 Agendas

An "agenda" is simply a prioritized queue of items where retrievals from the agenda are based on an agenda item's priority, or "weight." AdaTAU uses agendas in a couple of ways to coordinate the flow of control while inferencing. An agenda is used to keep a prioritized list of questions to be presented to the user during the interactive portion of AdaTAU inferencing cycle. Assigning a weight to questions and managing the questions with an agenda allow the modeler to control which questions are asked first. Since the answers to questions place new facts in the fact base, this order can be important. Questions are discussed in section 5.2.8. AdaTAU also uses an agenda to manage a weighted list of suggestions about which partitioned inference base should be inferenced in next. These suggestions are placed on the focus suggestion agenda by FRules (section 5.2.11). Additional attempts to queue an item already on an agenda increase the queued item's weight by the weight of the new item being added. No item appears in an agenda more than once. This

serves to increase the item's importance in the list.

5.2.5 Rules

A "rule" is a formalized statement that prescribes how a fact base can be changed based on the current state of the fact base. A very common style in which to specify rules is an "*if A, then B* " format where *A* and *B* are placeholders for one or more facts. For example, "*if* <watch, gold> *then* <cost, expensive>" Such a rule can be said to be "primed" if the facts in the collection *A* are all currently within the fact base. One possible action within a rule base system is to add all the facts within *B* to the fact base, after a rule becomes primed. Such a rule is said to "fire."

Those facts that must be present in the fact base before the rule is able to fire are called the "antecedent" facts. Such facts are also called "premises." Analogously, those facts which should be added to the fact base by the rule are called the "consequent" facts. Consequents are also called "conclusions." Thus, in the "*if A, then B* " rule formalism, *A* stands for the antecedent facts, and *B* stands for the consequents. In general, the lists *A* and *B* can be broken up further. For example, a rule could state that some of the facts in the *A* list should be absent in order for the rule to fire. Such facts would be "negative" antecedents. Analogously, some of the facts in the collection *B* might be identified as facts to remove from the fact base when the rule is fired. Currently, AdaTAU supports rules of the latter nature but not the former (see section 5.1.1).

5.2.6 Rule Bases

A complete collection of rules organized to capture knowledge in a particular domain is called a "rule base." A typical use of a rule base system is to begin with a collection of facts as well as a collection of rules, and then fire the primed rules successively, thereby causing new facts to be added to the fact base (or old facts removed). There are scheduling problems (for example, how to choose which of several primed rules to fire first) and this direct approach of incrementally building up the fact base is not appropriate in all cases.

Rather than providing a single kind of rule base, AdaTAU provides three kinds of rules, each contained in a corresponding rule base, and each supporting a different kind of forward-chained inference processing relative to a common fact base. These rules and described in the following sections. Other kinds of rules and rule bases may be added in future versions of AdaTAU. For example, action rules could be defined so application operations (actions) could be invoked with modification of the fact base depending on the results of the operation. Such a mechanism would enable the application to control and interact with the TAU process.

5.2.7 Irules

An "IRule" (Inference Rule) is a rule which directly affects the fact base, and requires no input from the user. IRules are the direct realization of "*if A, then B* " kinds of rules. An IRule's definition includes a list of the antecedent facts, a list of the consequent facts, and an optional justification string which can be used to document the purpose of the IRule in the inferencing process. When an IRule's antecedent facts are in the local fact base, the IRule is primed, and it will fire in the

Think phase of the AdaTAU inferencing mechanism (see section 5.3). The action of a IRule firing is to add the facts in the consequent list to the local fact base.

Errors occur if an IRule tries to add a fact to the fact base in a way inconsistent with that facts definition in the fact base schema. For example, if a fact is of the type **one_of**, then an IRule cannot add an attribute-value pair for that fact to the fact base if another pair is already in the fact base. This would contradict the fact base schema's definition allowing only one value for that fact attribute. IRules also cannot remove a fact as directed in the consequent list unless that fact is indeed in the fact base. An IRule's addition or deletion of facts in the fact base as a result of being fired must always agree with the inference base's fact base schema.

5.2.8 Questions

Questions are an important part of the AdaTAU rule-based inferencing process. They are tightly linked with QRules which are described in the next section. The question structure consists of the text of the question (which usually includes introductory information and the question itself), the question type, and a list of responses. Each response consists of a string which is presented to the user as a choice for answering the question, and a list of facts to be added to or deleted from the fact base if that response is selected. The two types of questions supported are **one_of** and **some_of**. Questions of type **one_of** allow exactly one response to be selected from the choices presented. Questions of type **some_of** allow at least one choice to be selected.

Questions are placed on an agenda as the result of a fired QRule. The agenda then presents the questions to the user according to the weight with which it was queued, with the heaviest weighted question being presented first. Questions are presented during the *Ask* phase of the AdaTAU inferencing mechanism (see section 5.3).

When a question has been presented and one or more choices have been selected, AdaTAU will then attempt to add the fact list associated with the selected response(s) to the local fact base. As with IRules, and facts added to the fact base or deleted from it must be consistent with the inference base's fact base schema.

5.2.9 Qrules

A "QRule" (Question-asking Rule) is a rule which involves the eventual processing of user input. Depending on a response to a particular question associated with the rule, other facts can be added to, or deleted from, the fact base. A QRule's structure consists of an antecedent fact list, a reference to a question associated with the QRule, the weight with which the question will be put on the question agenda, and an optional justification string which can be used to document the purpose of the QRule in the inferencing process.

Before a question associated with a QRule is posed to the user, the antecedent facts of the QRule must be present in the fact base. This causes the QRule to fire, and its resulting action queues the question associated with the QRule on the question agenda with the weight specified in the QRule's definition. AdaTAU separates the scheduling of the question, and the asking of the question and provides for the ranking of the question numerically when it is inserted on the agenda of questions to be asked. In this way the user is presented with the most important question first.

Type	Import and Export Semantics
`optional`	this fact is passed if it is present
`mandatory`	this fact is always passed and gets its value from the fact base
`focal`	this fact is always passed and gets its value from the FRule

Table 3: AdaTAU Fact Parameter Types

5.2.10 Fact Parameters

Every inference base may contain "fact parameters" which define the interface between that inference base and others in the partitioned inference base scheme. Fact parameters include "imports" and "exports," which are both lists of facts defined in the local fact base schema along with designations describing the facts' interface qualities. The imports list is a list of facts that the inference base will attempt to gather as parameters when it is first entered. The exports list is a list of facts the inference base will gather and return when the inference base is left as the result of following a focus switch suggestion (see section 5.2.11) or when it has finished inferencing.

A fact parameter in either the imports or exports list can have one of three types. These are `optional`, `mandatory`, or `focal`. Table 3 shows the semantics associated with each of these types of fact parameters. Focal parameters are different from the other two in the fact that the value passed for fact parameters of this type are taken from the export list of an FRule rather than from the local fact base or the fact base passed in when an inference base is first entered.

5.2.11 Frules

An *FRule* (Focus-suggestion Rule) is a rule that identifies an inference context (typically the name of another inference base) where it is likely that the goal of the current inference process will be served (i.e., additional facts can be deduced). An FRule's definition includes a list of antecedent facts, a list of facts to export, the name of an inference context where it is suggested inferencing should continue, a weight for this suggestion when it is added to the focus suggestion agenda, and an optional justification string which can be used to document the purpose of the FRule in the inferencing process.

An FRule does not itself derive any new facts, directly or indirectly, but it does supply a list of facts to be transferred to the next inference context when the FRule's focus suggestion is taken. When an FRule's antecedent facts are present in the fact base, the FRule will fire, and as its action cause the focus agenda to be modified. Depending on the weight attached to a focus suggestion, an FRule may lead to the suspension of inference within the current context, or it may direct attention to an alternate context when no further inference progress is possible in the current context.

When a focus suggestion is taken, the list of facts to export in the FRule that queued the focus suggestion is exported according to the fact parameter definitions for the inference base. An interesting feature of an FRule's export fact list is that facts can be identified by attribute only, which causes that fact to be exported with whatever value it happens to have in the local fact base. Facts exported when a focus suggestion is taken must be consistent with the local fact base schema and the fact parameter definitions.

5.2.12 Inferencers/Inference Bases

An "inferencer" is the AdaTAU construct which knits all the parts of an inference base together. The inferencer definition can contain the names of the inference base's IRule, QRule, and FRule rule bases. The definition of an inferencer also supplies the name of the inference base which is used by outside applications and other inference bases to identify the inference base. Because of this, the terms "inferencer" and "inference base" are often used interchangeably.

The definition of an entire inference base must have at least a fact base schema, an initial fact base (which may be empty), and an inferencer definition (which may identify no rule bases). It can optionally contain fact parameter definitions and one or more rule base definitions. (Note that a QRule base definition must necessarily include a question base definition.) The detailed syntax for encoding all these objects in the Rule Base Definition Language (RBDL) appears in section 6.

5.3 The Inferencing Mechanism

TAU is an acronym made up of the first letters of the phrase that summarizes the organization of this component — *Think, Ask, Update. Think* refers to the analysis of the fact base which is used to record information about the domain under consideration, and to the application of rules which directly modify this fact base (add or delete facts for example). This phase includes the firing of IRules and QRules. Rules considered during this phase may also lead to the scheduling of queries which will be processed subsequently. *Ask* denotes the capability of posing questions, and recording responses in answer to the questions, that are scheduled as a result of the *Think* phase. Finally, the *Update* phase will modify the fact base in a manner that depends on the responses recorded in the *Ask* phase.

AdaTAU has been enhanced beyond this inferencing scheme with the idea of partitioned inference bases. To implement this capability, another phase has been added to the process which can be thought of as *Think_After.* This phase applies the rules which queue suggestions to switch inference bases, or "focus." This is when FRules are fired. The next few sections will detail each phase of the AdaTAU inference cycle.

The phases of AdaTAU continue in a *Think, Ask, Update, Think_After* cycle until no new rules can be fired and there are no questions remaining to be posed and no focus suggestions pending. At this point inferencing stops and control is returned to the application which invoked the AdaTAU inferencing process. In the context of RLF reuse library advice, once there is no more advice to be given, the questioning of the user ceases and the application returns. It positions the user at the category or object associated with the last inference base that was inferencing. In this way, as inferencing control is transferred among the partitioned inference bases of the library advice, the user is conceptually navigated through the library model along the path of categories or objects associated with each inference base.

5.3.1 Think

An initial invocation of AdaTAU will process all IRules until no further changes to the fact base are possible. IRules will be examined in an arbitrary order; in particular, the rule base designer cannot assume any particular ordering of their being fired. The same arbitrary ordering is followed

in all subsequent passes through the list of IRules. During the *Think* phase, several passes through the set of IRules may be necessary since the addition of facts in the consequent lists of fired IRules may cause other IRules to become primed.

Then a single pass over all of the QRules is made so that all of these rules found to be primed can have their associated questions placed on a local agenda that is used to manage an orderly and prioritized interaction with the user. The examination of QRules will also occur in some fixed sequential manner. Multiple passes through this rule set is not required since these rules do not directly affect the fact base. After all the other phases are completed, the *Think* phase is invoked again because the fact base can be changed during the *Update* phase.

5.3.2 Ask

The user of a TAU-based application must be consulted when no further progress can be made within the *Think* phase. At this point, the agenda is consulted and a user's response to a question drawn from the agenda is processed. Question-asking and response-recording is handled by the *Ask* module. Other agenda items, if any, are not processed until after the next Update phase and the following *Think_After* and *Think* phases are completed. If the agenda is empty initially, and the Think phase does not add any items to the working agenda, the current AdaTAU invocation proceeds to the *Think_After* phase.

5.3.3 Update

From the recorded response returned by the *Ask* module, updates to the current fact base are handled by the *Update* phase. *Update* provides a truth maintenance phase. If a question was asked, depending on the response, consequences traced to the corresponding QRule are processed against the fact base. In the simplest case where no fact deletions occur, the *Update* phase simply needs to add those consequent facts attached to the particular response obtained from the user. Otherwise, *Update* must make sure that fact deletions are propagated through a fact dependency table that tracks the origin of facts in the fact base.

5.3.4 Think After

Think_After scans the FRule base and fires any FRule's that may be primed. The result of a fired FRule is to queue a "focus switch suggestion" on the focus suggestion queue. When an FRule fires it indicates that certain facts are present which suggest that inferencing may be more productive at a different one of the partitioned inference bases.

After any primed FRules are fired, the focus suggestion agenda is accessed. If it contains suggestions, the weightiest suggestion is taken and inferencing control transfers to a new inference base. The occurrence of a focus switch to a new inferencing context includes the exportation of facts from the current fact base and the importation of facts to the local fact base of the new inference base. This transfer is accomplished through the fact parameter facility. Fact parameters ensure that information gathered at one inference context can be used at the next one without having to recompute the information.

The rule base invocation strategy is designed to permit the application to start/suspend/resume separate TAU interactions using the individual rule base components. A key feature of this strategy is that these local TAU interactions will not be "greedy;" that is, exhausting all possibilities locally before considering TAU components elsewhere in the system. Instead, the system will operate on a "willing surrender" strategy that permits the controlling application to expect context switches after a single pass through the *Think, Ask, Update, Think_After* sequence.

6 The Rule Base Definition Language (RBDL)

This section introduces and describes the Rule Base Definition Language (RBDL). RLF library model advice modules are created by editing inference base specifications in RBDL and translating them to build the AdaTAU structures required. Although it is possible to simply write a program which instantiates AdaTAU via a sequence of calls to the AdaTAU ADT packages, this procedure is very difficult and does not correspond well to the inference bases being constructed. RBDL provides a method for fully defining one of a system's partitioned inference bases within the scope of one text file. Syntax exists in RBDL to build all the important AdaTAU constructs. RBDL facilitates description of the reuse library's library advice domain and hopefully makes the specifications easily maintainable.

Besides providing a convenient and readily modifiable medium for defining models, RBDL also promotes the reuse of inference bases or parts of them as assets in their own right. Since RBDL descriptions are ASCII text, AdaTAU inference bases described via RBDL specifications can be transported "as is" to any site. Also, since RBDL defines the separate inference bases of a partitioned inference base system separately, different collections of inference bases should be easily composable. Knowing the name of an inference base and seeing its fact parameter definitions is sufficient to tie that inference base into a set of partitioned inference bases.

The RBDL language and translator are implemented using SSAGS [PKP+82], a Paramax-proprietary meta-generation system based on ordered attribute grammars. RBDL may evolve in the future if better rule-based representations become predominant or the underlying AdaTAU technology is improved. Modelers will be able to use this language, not only to prototype library advice for reuse libraries, but to explore other rule-based strategies for reuse library management.

6.1 Representation of AdaTAU Entities

This section presents the language syntax for each of the library model entities described in section 5. It also shows how library advice modules, or "inferencers," are attached to the library model in a RBDL specification. See section 5 for more on inferencers. Syntax is presented in EBNF (Extended Backus-Naur Form). This is the same notation used throughout the Ada Language Reference Manual (LRM) [Ada83]. A brief description follows. A more complete description appears in section 1.5 of the Ada LRM.

lower_case_word
: nonterminal (e.g. inference_base_spec).

*italicized_part*_lower_case_word

refers to same nonterminal as the lower case word without italicized part. The italicized part is used to convey some semantic information (e.g. *fact*_identifier).

bold_face_word
language token (e.g. **fact**, ( ).

{item}
braces enclose item which may be repeated zero or more times.

[item]
brackets enclose optional item.

item1 | item2
alternation; either item1 or item2

A language syntax for RBDL without additional annotations is located in appendix B of this manual.

6.1.1 Facts and Fact Base Schemas

Facts and fact base schemas are the main substance of an AdaTAU inference context. Facts represent any information in the system and the fact base schema dictates the allowable values for all the facts in the inference base. Fact base schemas must conform to the following syntax:

factbase_schema_def ::=
fact base schema *fact_base_schema*_identifier **is**
fact_schema_def {fact_schema_def}
end [*fact_base_schema*_identifier] ;

fact_schema_def ::=
attribute_name_list : attribute_type
[attribute_value_list] ;

attribute_name_list ::= attribute_name {, attribute_name}

attribute_type ::= **some_of | one_of | any | reference**

attribute_value_list ::= (attribute_value {, attribute_value})

attribute_name ::= *fact*_identifier | string_literal

attribute_value ::= identifier | string_literal | number

This syntax shows that the fact base schema contains a list of one or more fact definition, called a "fact schema." Each fact schema contains a fact's name which may be an Ada-like identifier or

string literal, the fact's type, and a list of possible values the fact attribute can have. Fact values can be identifiers, string literals, or numbers.

An example of a fact base schema from an inferencer attached to the sort and search algorithms example library follows:

```
fact base schema schema_sort_algorithms is

   prior_context : one_of (algorithms, sort_algorithms,
                           internal_sort, external_sort, system, unknown);
   answer : one_of
      (internal_sort, external_sort, dont_know, still_dont_know, unknown);
   continue_confirmed : some_of (yes, no);
   no_hits : one_of (yes, no);

end schema_sort_algorithms;
```

6.1.2 Initial Fact Bases

An inference base's initial fact base is what fuels the first inferencing that takes place. The initial fact base often primes at least one rule so that the inference will begin the AdaTAU inferencing cycle. The only other way inferencers will begin inferencing is to pass them fact parameters which prime at least one rule.

The RBDL syntax for defining the initial fact base follows:

initial_factbase_def ::=
initial fact base *initial_fact_base*_identifier **is**
fact_list ;
end [*initial_fact_base*_identifier] ;

fact_list ::= **null** | fact_def {, fact_def}

fact_def ::= (attribute_name , attribute_value)

This syntax shows that initial fact bases are simply lists of attribute-value pairs which specify facts. These pairs must be consistent with the fact base schema.

An example of an initial fact base that follows comes from an inferencer attached to the search and sort algorithms example library:

```
initial fact base init_facts_algorithms is
   (answer, unknown),
   (continue_confirmed, no),
   (prior_context, system);
end init_facts_algorithms;
```

6.1.3 Rule Bases

This section presents the syntax for describing the three types of rule bases supported in AdaTAU. These rule bases are collections of IRules, QRules, or FRules. A QRule base requires a question base to be defined which supplies the questions referenced by the QRules.

IRules

IRule support basic "*if A, then B*" thinking and do not require any user interaction. An IRule can add facts to or delete facts from the local fact base. The RBDL syntax for an IRule base follows:

IRule_base_def ::=
irule base *IRule_base*_identifier **is**
IRule_def {IRule_def}
end [*IRule_base*_identifier] ;

IRule_def ::=
irule *IRule*_identifier **is**
antecedent : antecedent_fact_list ;
consequent : consequent_fact_list ;
[justification]
end irule ;

antecedent_fact_list ::=
fact_def {, fact_def}

consequent_fact_list ::=
fact_def |
neg_fact_def |
consequent_fact_list , fact_def |
consequent_fact_list , neg_fact_def

justification_def ::=
justification : text_block ;

neg_fact_def ::= ~ fact_def

This syntax points out that an IRule requires a list of antecedent and consequent facts, and can have an optional justification. An IRule base must contain at least one IRule.

Also of note is the syntax for a "negative" fact; that is, a fact that should be removed from the fact base, or "negated." When a fact is added to a fact base it is "asserted." The tilde character (~) shows that the fact following should be negated.

The following is an example IRule base from an inferencer attached to the Ada/X subsystem example library:

```
irule base widget_irules is
```

```
irule has_multiple_interests_1 is
    antecedent : (interest, ada_type), (interest, package);
    consequent : ~(multiple_interests, no), (multiple_interests, yes);
    justification : {Set multiple interests flag so we know to come
                     back for more.};
end irule;

irule has_multiple_interests_2 is
    antecedent : (interest, resources), (interest, package);
    consequent : ~(multiple_interests, no), (multiple_interests, yes);
    justification : {Set multiple interests flag so we know to come
                     back for more.};
end irule;

end widget_irules;
```

QRules and Questions

QRules and the questions that they queue on the question agenda are how AdaTAU conducts an interactive session with the reuse library user looking for library advice. The questions are collected in a question base and the QRules are collected in a QRule base which refers to the question base.

The RBDL syntax for QRules, questions, QRule bases, and question bases follows:

question_base_def ::=
question base *question_base*_identifier **is**
question_def {question_def}
end [*question_base*_identifier] ;

question_def ::=
question *question*_identifier **is**
text : text_block ;
type : attribute_type ;
[possible_responses]
end question ;

possible_responses ::=
responses : response_list

response_list ::= response {response}

response ::=
response_display {| response_display} =
consequent_fact_list ;

response_display ::= string_literal

QRule_base_def ::=
qrule base *QRule_base*_identifier
(*question_base*_identifier) **is**
QRule_def {QRule_def}
end [*QRule_base*_identifier] ;

QRule_def ::=
qrule *QRule*_identifier **is**
antecedent : antecedent_fact_list ;
question : *question*_identifier ;
weight : rule_weight ;
[justification]
end qrule ;

rule_weight ::= number

Of interest in question and QRule syntax is that more than one question response can share the same consequent fact list using the "or" bar (—). Also the question and question base identifiers in QRule and QRule base definitions, respectively, must match those specified in the question and question base definitions.

An example question and QRule base is taken from an inferencer attached to the sort and search algorithms example library:

```
question base questions_sort_algorithms is

    question sort_type_selection is
       text : { Select type of sort algorithm, internal for sorting an
                        array for example, external for data on tape.};
            type : one_of;
       responses :
          "Internal" => ~(answer, unknown), (answer, internal_sort);
          "External" => ~(answer, unknown), (answer, external_sort);
          "Don't know" => ~(answer, unknown), (answer, dont_know);
    end question;

    question clarify_question is
       text : { To determine whether the entire quantity of data may
                be sorted internally the size of available memory
                and the data space needed must be known.  Internal
                sorts are those which are done by sorting an array
                of structures for example.  External sorts involve
                tape files because of the large quantity of data.
                Select which type you would like to examine if you
                can.};
            type : one_of;
       responses :
```

```
            "Internal" => ~(answer, dont_know), (answer, internal_sort);
            "External" => ~(answer, dont_know), (answer, external_sort);
            "Don't know" => ~(answer, dont_know), (answer, still_dont_know);
        end question;

        question give_up_question is
           text : { There is no further advice I can give without a selection
                                  at this point.};
                type : one_of;
           responses :
              "Confirm" => (no_hits, no);
        end question;

end questions_sort_algorithms;

qrule base qrules_sort_algorithms (questions_sort_algorithms) is

        qrule sort_type_selection is
           antecedent : (answer, unknown);
           question : sort_type_selection;
           weight : 1;
           justification : {Determine whether or not internal or external
                sorting algorithms are desired.};
         end qrule;

        qrule clarify_question is
           antecedent : (answer, dont_know);
           question : clarify_question;
           weight : 1;
           justification : {To give an explanation and prompt for a choice
                again.};
         end qrule;

        qrule give_up_question is
           antecedent : (answer, still_dont_know);
           question : give_up_question;
           weight : 1;
           justification : { Cannot proceed without further input.};
         end qrule;


end qrules_sort_algorithms;
```

FRules

FRules are the rules which allow AdaTAU to conduct a distributed inferencing over partitioned inference bases. Although they do not affect the local fact base, they queue focus switch suggestions which contain the names of other inference bases where additional inferencing should occur. The

RBDL syntax for FRules and FRule bases follows:

FRule_base_def ::=
frule base *FRule_base*_identifier **is**
FRule_def {FRule_def}
end [*FRule_base*_identifier] ;

FRule_def ::=
frule *FRule*_identifier **is**
antecedent : antecedent_fact_list ;
export : export_fact_list ;
focus : *inferencer*_identifier ;
weight : rule_weight ;
[justification]
end frule ;

export_fact_list ::= fact_list

Of interest, the export fact list of an FRule specifies the facts that will be exported if the FRule's focus switch suggestion is taken. This list may include facts identified by their attributes only. This allows facts which may have undetermined values at the time of the focus switch to still be exported as fact parameters. The fact parameter definitions detail how these facts are exported. Also, it is important that the inferencer identifier in the focus field of an FRule identifies an inference base that has been built.

An example FRule base from the sort and search algorithms example library advice:

```
frule base Sorting_Frules is

        frule Internal_Interest is
                antecedent : (answer, internal_sort);
                export : (prior_context, sort_algorithms);
                focus : internal_sorts;
                weight : 1;
                justification : {Since advice on internal sorting algorithms
                        is desired, we will move there.};
        end frule;

end Sorting_Frules;
```

6.1.4 Fact Parameters

Fact parameters are as important as Frules in allowing AdaTAU to support partitioned inference bases. These define which facts can be passed into and out of an inference base when a inferencing focus switch occurs. The RBDL for fact parameter definitions follows:

fact_parameters_def ::=
fact parameters is
[import_list] [export_list]
end fact parameters ;

import_list ::= **imports : (** param_description_list **) ;**

export_list ::= **exports : (** param_description_list **) ;**

param_description_list ::=
param_description {, param_description}

param_description ::=
*fact*_identifier => **optional |**
*fact*_identifier => **mandatory |**
*fact*_identifier => **focal**

It is important to make sure that facts will be in the local fact base or supplied by an FRule when those facts are listed as fact parameters. Errors can be caused by making fact parameters the wrong type or trying to export or import fact that will not be available.

Here are some simple fact parameters from an inferencer attached to the sort and search algorithms example library:

```
fact parameters is
        imports : (prior_context => mandatory);
        exports : (prior_context => focal);
end fact parameters;
```

6.1.5 Inferencers/Inference Bases

An inference base definition is usually the entire contents of a file. This allows a certain modularity to library advice construction. The inferencer RBDL construct collects all the parts of the inference base that are available to the forward-chaining inference engine. The syntax for inferencers and inference bases follows:

inferencer_def ::=
inferencer *inferencer*_identifier **is**
[IRule_base_specification]
[QRule_base_specification]
[FRule_base_specification]
end [*inferencer*_identifier] **;**

IRule_base_specification ::= **irule base :** *IRule_base*_identifier **;**

QRule_base_specification ::= **qrule base :** *QRule_base*_identifier **;**

FRule_base_specification ::= **frule base :** *FRule_base_*identifier ;

inference_base_spec ::=
factbase_schema_def
[fact_parameters_def]
initial_factbase_def
{rule_base_definition}
inferencer_def

An inferencer need not have any rule base definitions. This is usual while prototyping to create a placeholder inference base. An inference base must have a fact base schema, an initial fact base (which may be empty), and an inferencer definition. It optionally can have a fact parameters definition and one or more rule base definitions.

An example of an inferencer definition from an inferencer attached to the sort and search algorithms example library follows. No example inference base is given since it consists of a whole file and may be examined using a text viewer.

```
inferencer sort_algorithms is

        qrule base : qrules_sort_algorithms;
        frule base : sorting_frules;

end sort_algorithms;
```

6.2 Other Syntax

The syntax for various low-level, RBDL primitives follows. These primitives may be mentioned in the syntax examples above and are included for completeness. A full syntax summary for RBDL is given in appendix B.

identifier ::= letter {[underline] letter_or_digit}

letter_or_digit ::= letter | digit

letter ::= upper_case_letter | lower_case_letter

number ::= digit {digit}

text_block ::= { any_letter_but_bracket {any_letter_but_bracket} }

7 Using the Language Translators

The RBDL or LMDL specifications are prepared by creating them with a system text editor. The modeler should be careful to obey the syntactic rules of the language so that the language translator can smoothly translate the specification and build the associated knowledge base representation. However, if a mistake is made in a specification, an error message is reported on standard output with a line number indicating the approximate location of the offending statement. The language translators are currently limited in the way such errors are reported and a new user may find that only the first error is reported even when the specification contains several such errors.

If no syntax errors are detected, the language translators, `Lmdl` and `Rbdl` perform some semantic error checking. The RBDL translator currently will perform only token semantic error checking. The LMDL translator performs more extensive semantic checking. Any semantic errors that either translator misses should be caught at run-time within the appropriate routines which construct the knowledge base representations.

The RBDL or LMDL language translators are invoked by simply typing:

```
Lmdl file.lmdl
Rbdl file.rbdl
```

or

```
Lmdl -input file.lmdl
Rbdl -input file.rbdl
```

or

```
Lmdl < file.lmdl
Rbdl < file.rbdl
```

where `file.lmdl` or `file.rbdl` is the name of the file containing the LMDL or RBDL specification, respectively, as constructed in a system text editor.

7.1 Command Line Options

There is also a family of command line options which can be used with the language translators to tailor its execution. The options are:

```
-help              prints available command line arguments
-I <pname>         uses RLF libraries found in the directory
                   specified by pathname <pname>
-d                 enables debug messages from the RLF tools
-w                 enables warnings for unknown options
-q                 quiet mode - don't output build messages
-model             build library model hierarchy only
-state             bind attributes and inferencers only
-input <fname>
<fname>            specify name of file to translate
                   (-input is optional and input defaults to stdin)
```

Building just the library model structure parts of a specification includes all definitions except attribute definitions at categories or objects. Binding just the attributes will only make associations between files, string, integers, and also inferencers and the categories and objects where they are defined. This is very useful when changes to the library model only affect the attributes since `Lmdl` will execute much faster without having to build the entire library model structure.

7.2 Using the `.rlfrc` Start-Up File

RLF 4.0 also supports the ability to set some global operation parameters by reading a start-up configuration file. This file is named `.rlfrc`. When an RLF application begins execution it will search the current working directory and then the directory identified by the `HOME` environment variable to find a file named `.rlfrc`. It will then read the first file it finds with this name and set certain global variables for RLF based on its contents. If it does not find a file named `.rlfrc` in either of these locations, it will skip reading.

A few of the global operating parameters that can be set in the start-up file configure the RBDL and LMDL language translators. Both translators can be given a file name from which to read the specification to translate or be set to run in quiet mode, not outputting any build messages. Additionally, the LMDL translator can be set to only translate the library model structure or bind the attribute attachments ("state") of a LMDL library model specification. Setting these parameters in the `.rlfrc` file uses the following syntax:

translator_setting ::=
    translator : translator_type

translator_type ::=
    lmdl : lmdl_setting |
    rbdl : rbdl_setting

lmdl_setting ::=
    quiet_translation | translate_only | default_input_spec

rbdl_setting ::=
    quiet_translation | default_input_spec

quiet_translation ::=
quiet : flag_setting

translate_only ::=
only : model_or_state

model_or_state ::=
model | state

default_input_spec ::=
default specification : pathname

For example, the following lines, when placed in the `.rlfrc` file, would make both translators run in quiet mode, have `Lmdl` only translate the library model structure parts of the input LMDL specification, and have `Rbdl` run on the file named `test_inferencer.rbdl` by default.

```
--|
--| Specification translator settings
--|
translator: Lmdl: quiet: no
translator: Lmdl: only: model
translator: Rbdl: quiet: no
translator: Rbdl: default specification : /path/models/test_inferencer.rbdl
```

A complete syntax summary and example `.rlfrc` file appropriate for most day-to-day RLF operations can be found in appendix D.

8 Creating Library Models with LMDL

8.1 Hints for Modeling Libraries

This section provides some hints for constructing efficient and understandable library domain models using LMDL. This section does not present a formalized method for modeling with LMDL, but rather presents some experience gathered from previous efforts to create library models with LMDL. While the basic principles of RLF's domain model approach are presented in section 2, and AdaKNET and LMDL are described in sections 3 and 4, this section will present some practical information about developing library models with RLF.

The library domain being modeled will have great impact on the final form of the library model specification. Most reuse libraries built using RLF will grow down from the application domain being modeled and up from the existing set of reusable assets to be made available in the library. This sort of two-way approach, filtered through the organizational preferences of the domain modeler, can produce library model specifications, and subsequently RLF reuse libraries, with very different feel and structure. LMDL and RLF are flexible enough to support varied modeling strategies, but the highly-structured nature of a library model specified in LMDL should produce libraries with enough similarity of structure that they can be reused by other modelers, administrators, and users.

8.1.1 Basic Structure of a Library Model

The encoding fundamentals of LMDL and the AdaKNET structured inheritance network subsystem providing the underlying representation imply a certain minimal, basic organization to a library domain model. Every library model necessarily has a root category which is the most general class of things described in a library model. Traditionally, a root category named `Thing` has been used as the most general description, but other names are allowed for the root category. Whatever name is chosen for the root category, care should be taken to make sure that any other category descriptions which occur in the library describe subsets of the the set of things described by the root category.

Often when a library model is developed, initial effort centers around building a structure which directly addresses the description of the existing or expected reusable assets which the library is making available. This is often called the "main sub-model", or just the "main model." This term reflects the fact that the library model centers around the assets in the reuse library.

A sketch of the main model using the specialization hierarchy to form the general structure is usually easily produced first. Definitions of categories in the main model are then expanded and refined using relationships to better describe and differentiate the categories and objects. It is often necessary to introduce new categories which will be referenced as the types of these relationships. These categories often model things which do not fit in the main model hierarchy of categories. It is useful to create a first-level subcategory of the root category which serves as the parent of these relationship filler types. This category is probably most appropriately named `"Relationship Values"` or `"Relationship Types"`, but any representative name would be good. For example, in the sort and search algorithms example library model, this category is named `"Attribute Values"` since its child categories provide filler type values for relationships which describe the categories in the main model.

Another essential part of any library model is the "action sub-model." The action sub-model describes all of the RLF actions which are defined at categories and objects elsewhere in the library model. Certain parts of the action sub-model must exist in every RLF library model which wishes to invoke actions. This is best described in section 3.3.6. The `"Action Definition"` category which is the root of the action sub-model is often best included as a first-level subcategory of the library model's root category.

A library model development template which can be used as a start for domain model encoding with LMDL is provided with RLF. It shows an example of the common model organization used above and can be found in appendix C.

8.1.2 Depth of Detail in Library Model

Once the RLF modeler has worked enough with LMDL, it will become apparent that the most fully described and precise library model would include category and object descriptions of everything in the world. This, of course, is not a practical goal when modeling an application domain. Subsequently, the level of detail modeled in a library model is an open question.

Structuring the reuse library with a domain model is only effective if enough information is provided in the model to distinguish between similar assets both modeled as objects in the library model. This should be a guide as to what is an acceptable level of detail in the library model. Providing more

detail than this might provoke the user to consider aspects of the assets that were not considered before, but too much detail can swamp the asset descriptions in so many other categories and relationships that they are difficult to find in the over-structured tangle. Less than distinguishing detail may be warranted if the model is only to be used for education or understanding of a particular domain. In this case, even object modeling may not be necessary.

Another issue regarding depth of detail in the library model is the threshold at which the model crosses from a description of the real world to the real world objects themselves. At what point should the description of a number leave off being a description and become an integer attribute? Should there be an object for every possible number which fills a relationship, or should the object "pseudo-fill" the relationship by supplying a local integer attribute to describe that quality? There are no clear answers to these questions, but a good guideline to follow is the importance of the descriptions on the overall model. If a description of numerical constructs is important to the library domain, then it may be worthwhile to do a little extra modeling of the numbers and supply objects for many of them. If numbers represent some descriptive quality of an object, e.g. size, but are not important in the whole model, then it is perfectly acceptable to attach integer attributes named **size** to the appropriate objects, and no number modeling may be required at all.

8.1.3 Using Attributes Effectively

The use of attributes in the library model is something to think about when developing a library model. Attributes play an important role in establishing the depth of the model as mentioned above, and they also are important for setting the threshold between description and the actual items being described. Attributes are the library model's connection to the actual assets modeled in the LMDL specification and available in the RLF reuse library, so some care should be taken when deciding this connection.

One thing to consider is the choice of whether to use string or file attributes when developing a library model. Using file attributes allows a degree of disconnectedness between the LMDL definition of the attribute and the attribute itself. The contents of the file specified in the file attribute definition can be changed and accessed from its attribute description in the library without having to re-translate the library model specification. This is why file attributes are useful for representing the actual reusable assets, since new versions can be put into the library simply by having the library administrator update the correct file. The files named in file attribute definitions, however, need to be managed so that using file attributes for every text attribute may not be a wise idea.

String attributes are useful for getting small text information into the library model descriptions. They have an advantage over relationships; they can be easily viewed and manipulated by actions when used as action targets or agents. They also avoid the need to have to do modeling of a **String** or **Text** domain with very many objects to contain all the text information describing categories and objects. The disadvantage of using string attributes is that the information contained in the string is modeled into the library model specification and will need to be updated there if the information changes. If a string attribute turns out to be overly large or likely to change often, then it is probably better modeled as a file attribute. The need to re-translate the specification after updating a string attribute is not necessarily very time consuming. Since only an attribute binding has changed, the specification can be re-translated doing attribute bindings only, which

takes significantly less time that translating the entire library model specification. See section 7 on how to translate a LMDL specification in this way.

8.2 Hints for Modeling Actions

The action mechanism in RLF is very flexible, and unfortunately, this can make the modeling of actions complex. The best way to model actions is to follow the examples in the starter library model template (appendix C) or the example library models, and to reference the section on adding actions in 10.2.2. Modeling actions entails modeling action categories in the action sub-model and modeling action definitions at categories or objects elsewhere in the model.

One hint in modeling action categories is to take advantage of the separation that invoking strings in an operating system shell can provide. The simplest "**System String**" actions pass one line of command to the shell, wait until it completes, and then RLF continues. Much more can be accomplished by using the operating system to the RLF's advantage. By ending the string attribute at the action category with an ampersand, **&**, the action will run in the UNIX "background" when the system string is executed. This allows the RLF application to continue running while long actions or actions that open their own windows are being invoked. Another trick is to have the "**System String**" action invoke a UNIX shell script. This way the shell script can do many things one line of command could not. A script could set environment variables, run several tools, record information in a file, and then return, for instance. PCTE issues regarding the modeling of actions, and action categories in particular, can be found in appendix E.

The library domain modeler should take advantage of the RLF semantics of actions as expressed in section 3.3.6 when defining actions at categories or objects. Inheritance, targets, and agents can all be used in combination to produce different effects. For example, suppose the library model defines a group of objects which are all children of the same category and all have a file attribute named **contents** which holds the asset described by each object. Then, a view action defined at the parent category could be inherited to each of the objects so that it need not be defined repetitively for each object. The action definition at the parent category should also define a target named **contents** which would be inherited. The **contents** file attribute need not exist at the parent category, since when a list of actions to invoke is requested by an RLF tool, if targets are specified and there are no attributes with matching names, then the action will not be returned as available. Using inheritance to reduce repetitive action definitions is a highlight of action definition in LMDL.

Use of action agents to make actions more flexible is powerful but sometimes complicated. Section 10.2.2 gives a good example of a general print action using agents. Agents are most useful when they can express action invocation dependencies for attributes attached to different categories or objects. For example, if a general print action is desired for all the assets described in a library model, then it could be defined high in the category hierarchy and inherited to all objects describing assets. A reuse library is likely to have assets of different forms, however, such as source code in ASCII, documents in PostScript, and perhaps X Window System bitmaps. One inherited print action which specified the local PostScript printer directly with no options would not work for all the assets it needed to. Using agents in the action category describing the print action to specify which printer and what options to use allows objects to define local string attributes which set these for that object's type of asset, and enables one print action to serve all assets. This avoids the propagation of special-purpose print actions for every kind of asset described in the library model.

Of course using action agents has a negative side, too. If too many are used to parameterize an action, then many string attributes will need to be defined at every object just so the action can be invoked correctly. This causes the number of string attributes to be unmanageably large and creates a lot of objects whose attributes will need to be updated if that action should change.

8.2.1 Using the Built-In Ada Procedure Actions

The built-in Ada procedure actions are included with RLF to provide a basic functionality for processing library assets which can serve as the default asset processing mechanisms for most simple libraries. Large libraries which limit access control or perform configuration management or other operations on assets when they are extracted may need to replace these built-in procedure actions with more complex actions of their own.

The built-in Ada procedure actions are made available to the categories and objects of a library model by including the appropriate action definition in the library model's action sub-model. These action definitions can be found in the example library model specifications and in the starter library model specification in appendix C. The built-in actions, once defined in the action sub-model, are accessed in the main library model by defining them at the category or object where they are desired. They are subject to inheritance and restriction just like any other type of action. The definition in the main library model should include an action name by which the action will be identified, and the action category name of the Ada action. This should be one of **`Extract`**, **`Import`**, **`Export`**, or **`"Display Attributes"`**. The built-in Ada procedure actions currently do not take gets or agents; they operate on all available attributes of the category or object where they are invoked as described in section 3.3.6. For examples of main model action definitions of the built-in Ada procedure actions, see the example library model specifications delivered with RLF.

8.3 Connecting Advice to the Library Model

The ideal library model would have a library advice module attached to every category and object. In this way, the library user could receive instructive information from the modeler about the library model from any place in the hierarchy. Realistically this is not possible, because for a large, descriptive model there will be too many categories and objects to support development of an inferencer for each one.

The best strategy for connecting inferencing advice modules is to attach them to the category hierarchy in the main model. The goal of most RLF reuse libraries is for the user to browse the knowledge model to find a desired asset to reuse, identify the asset, and extract it. Supplying inferencer support for the main model is then the best, minimal advice capability a library should have. The inferencers should help the user navigate to the area of the model of most interest to the user (as revealed through asking questions) and then help distinguish between different objects describing assets with similar descriptions.

It is important for the RLF modeler, when encoding inferencer attachments in LMDL, to remember that all inferencer names currently must be given in lower case. This is because the RBDL translator converts all RBDL specification identifiers to lower case when internalizing the information. For more information on modeling RLF library advice modules see sections 5, 6, and 9. The syntax for attaching inferencer to LMDL categories and objects is presented in section 4.

8.4 Debugging Hints for LMDL

Error messaging in LMDL for RLF 4.0 has been improved over the error handling of SNDL (which LMDL replaces) in previous versions of RLF. Syntax errors will report the line number where the error occurred and some context information about what the language parser was doing. Semantic errors print an error message, and then the translator will continue processing the specification for as long as it can. This allows the modeler to discover and correct multiple errors in one pass through the translator. One way to make the best use of the error messages generated by LMDL is to capture the LMDL translator's output in a file. This can be done using UNIX file output re-direction. Invoking the LMDL translator, `Lmdl`, as follows:

```
Lmdl file.lmdl >& LOG
```

will save the LMDL translator's output in a file named `LOG` when translating the LMDL library model specification in a file named `file.lmdl`. After this has been done, the file named `LOG` can then be surveyed to look for errors. This file is also useful to provide when asking for assistance as directed in section 11.

Most LMDL semantic errors will be caused by incorrect ranges on relationships in the library model specification. It is important to only restrict relationship ranges to be smaller than the range inherited for the relationship. Also, when differentiating relationships, either by subsets or partitions, it is important to make sure that the ranges provided for the subrelationships, if they were to be filled, still allow the relationship as a whole to match its range constraints. Subrelationships should be especially careful to allow the lower bound of the parent relationship's range to be correct when considering the lower bounds of all the subrelationships. The parent relationship's range can always be restricted before differentiating the relationship so that the subrelationships can have the desired ranges. Review section 3 frequently to check the semantics for different AdaKNET entities when doing complex modeling, especially relationship differentiation.

9 Creating Library Model Advice with RBDL

9.1 Hints for Modeling Advice

Modeling library model advice can become complex depending on the depth of knowledge represented and manipulated in the library advice modules. A minimal inferencer supplying library model advice may only present navigational choices based on the category hierarchy corresponding to where the inferencer is attached in the library model. More developed inferencers might tailor their questions and inferencing according to responses the user had made to earlier questions about the level of expertise of the user.

Inferencer development should center on capturing information about the assets in the library model or the structure of the model which may not be evident from using the RLF tools to examine the model. Inferencers allow the RLF modeler to encode many of the decisions that were made when constructing the library model. This information can then be supplied to the user when advice is requested.

Unfortunately, the current interface between the AdaKNET semantic network used to represent

the library model and the AdaTAU inferencing system which provides the library advice is very minimal. Therefore much of the information already modeled in the library model will need to be reworked into facts and rules which AdaTAU can process. This interface may be developed further in the future to allow AdaTAU to query the library model so that this replicated domain information can be kept to a minimum.

This section will not present a formalized method for the development of library advice modules using RBDL. Instead it will present some useful RBDL modeling expertise gathered from efforts developing inferencer advice support for RLF library models. Sections 5 and 6 should be consulted freely when developing inferencers since the inferencing cycle of rules processing facts may not be familiar to manby RLF modelers. The parts of section 2 discussing the library advice domain is also useful. Examples of inferencers supplied with the RLF example libraries will prove indispensable when developing new inferencers. Modeling support as described in section 11 is also available.

9.1.1 Practical Breakdown of Rule Usage

This section presents a little experience about the distribution of RBDL's different rule types in an average collection of inferencers. AdaTAU was originally constructed as a general Ada rule-based inferencing system, but has shown tendencies for certain distributions of the different rule types when in an RLF role providing library model navigation and expert advice.

IRules assert facts derived from other pre-existing facts. IRules may be the least useful in an inter-active library browsing approach. These would probably be more useful in a stand-alone distributed inferencing application.

QRules assert facts as a result of the answer to a question. QRules are the workhorse of AdaTAU. They allow the application to find out what the user really wants and what the application needs to know to continue inferencing. Typical library advice modules will be mostly composed of QRules.

FRules suggest which distributed inferencer to go to next based on pre-existing facts. FRules are essential in the library context because they cause the traversal of the AdaKNET to categories or objects thought to be more of interest to the user. Every library advice module will probably contain at least an FRule for each child category or object and possibly one or more additional FRules for the parent(s) and other categories or objects.

9.1.2 Selecting Facts

Selecting the actual facts which will be manipulated in the inferencing process can be one of the trickiest parts of designing an AdaTAU instance. After the library advice domain has been identified and refined, it is then necessary to encode the domain knowledge into a set of facts and the rules that operate on them. The proper selection of facts and their attributes and types will decide the complexity and usefulness of the inferencers providing expert advice for a reuse library.

Three groups of facts can often be identified when refining the library advice domain into an AdaTAU instance:

General system facts These facts will be set and read to help control the flow of inferencing between inferencers and to record general information about the goals of the user. For example, it might be worthwhile to keep a fact which tells you the name of the last inferencer called, or that the user is a programmer and not a manager.

Context-specific facts These are facts which are best set at the particular category or object associated with the inferencer. These fact model knowledge best obtained from the user or other facts when inferencing at a particular category. They help decide where to go next and set more information about the user's desires. For example, at the category named "`Sort Algorithms`", there should be facts detailing what kind of sort the user is interested in because this is the most logical place to ask. Lower in the category hierarchy, the user may already know which sort is needed, and if not then inferencing could be directed to the inferencer attached to the more general "`Sort Algorithms`" category.

Delayed facts These are a kind of context-specific fact which are set at one category then shuttled from inferencer to inferencer as fact parameters until the facts are actually needed. For example, it is probably best to ask the user if more than one thing is being sought, but this fact probably won't be useful until the user has found the first thing and would then like to look for the others.

Once it is decided what information is useful for the AdaTAU instance, this information should be split into a number of facts that will hold all the information. This is when the modeler should decide what type the facts should be (i.e., whether they should be `one_of`, `some_of`, or `any` facts). Facts of type `one_of` are best for holding flags, or information where only one value is expected at any time. For example, a fact called `power`, could have values `on`, `off`, or `unknown`, but should only be one of these at a time. Facts of type `some_of` are best for recording lists of information. For example, a fact called `best_colors` might have values `red`, `orange`, `yellow`, `green`, `blue`, and `purple`, and will probably be more than one of these. Facts of type `any` and `reference` haven't proved to be very useful when developing library advice modules. Facts of type `any` take on one arbitrary value and facts of type `reference` have a string as a value, e.g. a filename, which is a reference to the fact's actual value.

9.1.3 Partitioning and Encoding the Library Advice Domain

Once the list of fact attributes, values, and types is decided, they must be divided up into different inference contexts. This means deciding which facts should exist at which category/inferencer. A lot of the facts may exist at some or all the inferencers, and be passed from one to the other as fact parameters.

When deciding the import and export fact parameters for an inferencer, time should be spent trying to keep the number of fact parameters low and relevant. An inferencer should be as self-contained as possible with only important "general system", "delayed" or result facts being exported

or imported. "Delayed" or "general system" facts can often be shuttled (imported or exported) between inference bases by fact attribute only to reduce the number of Frules required and to reduce the complexity of the interface between inferencers.

Once all the facts are chosen and there is a rough sketch of which categories/inferencers they belong to, the inferencers must be encoded in RBDL and knitted together. Specifying the fact base schema and initial fact base portions of the RBDL specifications should come easily from the facts decided on for the inferencers. Coding the IRules, Questions, and QRules should be based primarily on which facts are considered "context-specific" for a particular category/inferencer. Coding the fact parameters and FRules should be based primarily on "general system" and "delayed" facts along with any useful facts produced as the consequences of the IRules and QRules.

9.1.4 Connecting the Inference Bases

Once the library advice domain has been encoded into RBDL inference bases, these bases must be constructed and attached to the AdaKNET representation of the LMDL library model. The inferencers are constructed by the RBDL translator whose operation is discussed in section 7. For the inferencer to be accessible from a category or object while running an RLF tool, an attachment statement must be provided in the LMDL specification for the library model. Attaching initial or additional advice modules is discussed in section 4 for the syntax and section 10 for the procedure.

NOTE: The names of inferencers expressed in inferencer attachment LMDL clauses must appear in all lower case. This is because the RBDL translator currently converts all identifiers in the inferencer's RBDL specification to lower case. The identifier may appear in mixed case in the RBDL specification, but will be converted to lower case internally.

9.2 Debugging Hints for RBDL

Debugging distributed inference bases encoded in RBDL can be difficult since the RBDL translator is not good at semantic error detection and reporting. Because of this, many errors in inferencer modeling appear as exceptions in the RLF tools when running the inferencers. Although this is true, there is some advice on locating common RBDL errors.

Most RBDL errors are the result of mismatched fact and fact attribute names. The RBDL translator does not ensure the all occurrences of facts in rules and fact parameters match the facts defined in the inferencer's fact base schema. Extra care should be taken to make sure that all instances of a fact in an inference base match the fact's definition in the fact base schema. If an occurrence of the fact does not match when an AdaTAU operation is being performed, this will often cause an error in the `Install` routine when the fact is being added to the local fact base, but does not fit any description in the fact base schema. So exceptions raised from this routine usually indicate an inconsistency in the names used in a fact-attribute pair.

If an exception is raised from the `Install` procedure while in a procedure named `Eval_Focus`, then this usually indicates a problem with fact parameters. `Eval_Focus` is the routine which engineers a switch between different inference bases after an FRule was suggested a context switch. All fact parameters specified for export in the fact parameters section of the RBDL inference base specification must have been asserted by a rule prior to any context switch.

If an exception is raised from the **Install** procedure while in a procedure named **Process_Response**, then this usually indicates that the consequent facts of a question are not correct. The question base should be checked to make sure the consequents are legitimate fact-attribute pairs as defined in the local fact base schema.

Another common error which can lead to exceptions when executing the AdaTAU inferencing engine is the attempt to assert a fact of type **one_of** which already has an existing value in the local fact base. Asserting a new value for a fact will not override a value already in the local fact base. The existing fact-attribute pair must be retracted before the new instance can be added. For example, suppose the fact base schema for an inference base from the sort and search algorithms library advice domain contains the following fact schema:

```
answer : one_of
   (searching_algs, sorting_algs, dont_know, still_dont_know, unknown);
```

Also suppose the initial fact base contains an initial value for this one_of type fact:

```
initial fact base init_facts_algorithms is
        (answer, unknown),
        (continue_confirmed, no),
        (prior_context, system);
end init_facts_algorithms;
```

Then, if it later becomes apparent through inferencing what kind of algorithm is desired by the user, then the already asserted value of the fact must be retracted before a new value is asserted. This is shown in the following question from the same inference base:

```
question sort_or_search_question is
   text : {Select whether advice is desired on algorithms to do
             sorting or searching, or select don't know for more
             information.};
   type : one_of;
   responses :
      "Sorting" => ~(answer, unknown), (answer, sorting_algs);
      "Searching" => ~(answer, unknown), (answer, searching_algs);
      "Don't know" => ~(answer, unknown), (answer, dont_know);
end question;
```

RLF modelers should take advantage of modeling support available as described in section 11. Domain model encoding and debugging in RLF is not always easy to understand or straight-forward. Modeling correct RBDL inference bases is especially hard to do without experience modeling RBDL. Any difficulties establishing library advice modules for a reuse library should not convince an RLF modeler to forsake using the advice mechanism, since it is an important advantage in using the domain modeling approach to organizing a library of reusable assets.

10 Domain Model Maintenance

This section outlines for the library modeler the methods for developing, modifying, and maintaining an RLF library domain model. The outlines make reference to running the LMDL translator and the examples appear in LMDL. LMDL is fully described in section4. Examples in this section assume that the RLF_LIBRARIES environment variable (further described in the **RLF Administrator's Manual**) has been set to the directory containing the reuse library being modified and that the LMDL translator, `Lmdl`, appears in the library modeler's path.

10.1 Building New Libraries

New libraries are constructed by running the LMDL translator on the LMDL specification of the library model. Assuming that `spec.lmdl` is the LMDL specification of the library model, the command

```
Lmdl spec.lmdl
```

issued at the UNIX shell prompt will construct the new library in the directory to which the environment variable **RLF_LIBRARIES** is set. If a library with that name already exists, it will be overwritten. Using the LMDL language translator is more fully described in section 7.

10.2 Modifying Existing Libraries

This subsection describes the various library modifications which a library modeler is likely to make. Most modifications require the editing of the library's LMDL library model specification. After this editing has occurred the specification must be retranslated with the LMDL translator, **Lmdl**. Assuming that the library's LMDL library model specification is in a file named **spec.lmdl**, and that the **RLF_LIBRARIES** environment variable specifies the directory where the library has been built, then running the LMDL translator is accomplished by issuing the following command at the UNIX shell prompt:

```
Lmdl spec.lmdl
```

When this command completes successfully, the library model specification has been "retranslated" and any changes made while editing the specification will now be realized when the library is viewed with an RLF application.

Retranslating is something the RLF library modeler is likely to do often to check on changes made while developing a LMDL library model specification. If the only changes made to the library model specification were changes to the integer, character string, or file attributes of categories or objects, the LMDL translator should be invoked with the **-state** command line option when retranslating. This significantly reduces the amount of time required to retranslate the library model specification. Also, for translation of large library models, the best performance is realized if the library model instances directory is located on the machine whose CPU is running the LMDL translator.

10.2.1 Attributes

Attributes are an important part of any library domain model. They are the RLF mechanism that allows a reuse library to bridge the gap from an abstract description of a reusable asset to the actual asset itself. Assets are represented in the library model by integer, character string, and file attributes of categories and objects. In most cases, assets will be file attributes of objects. This subsection describes how attributes can be manipulated to attach and remove real data to points in the library domain model. This should be a common process for an RLF modeler supporting an active reuse library.

Adding New Assets to the Library Model

A new asset is added by defining an attribute at the object which best represents the asset. Sometimes an object must be created to represent the asset if one does not currently exist. After these

changes, the LMDL specification of the library model must be retranslated for the new attributes and objects to become part of the library.

Qualifying an asset so that it is best described in the library model specification can be a complex task. With a good domain model design, most new assets to be modeled will already have a distinct area of the library model which describes them. It is not uncommon however that new parts of the library model category hierarchy need to be constructed to best fit the description of a new reusable asset. Sometimes addition of new assets will force re-thinking of some domain modeling decisions and cause changes to basic structure.

The usual way to include a new asset in a reuse library domain model is to create an object in the model at the point where the new asset is best classified. Suppose there is a reusable quick sort implementation which is to be added to the sort and search algorithms library. By browsing the library with the **Graphical_Browser** or the library model specification with an editor, the quick sort algorithm category is located and determined to be the most representative of the new asset. The quick sort category appears as such:

```
category Quicksort ( "Exchange Sorts" ) is
   restricted relationships
      has_best_case_of (1 .. 1) of Logarithmic;
      has_avg_case_of (1 .. 1) of Logarithmic;
      has_worst_case_of (1 .. 1) of Quadratic;
   end restricted;
   attributes
      file desc_source is "sort_and_search/exchange_sort_desc";
   end attributes;
end category;
```

First an object is created by editing the specification to include the following:

```
object "Example Quicksort" ( Quicksort ) is
end object;
```

Next the actual asset is attached to the object by defining a file attribute of the object. The file containing the asset is given a pathname relative to the directory **Text**, which is a first-level subdirectory below the directory where the library representations exist. From the modeler's point of view, the actual name and location of the asset may be the library administrator's concern. The important task for the modeler is to include the object describing the asset in the most logical place in the library domain model. Assuming that the asset file name and location are known, the new object definition would now look like this:

```
object "Example Quicksort" ( Quicksort ) is
   attributes
      file source is "sort_and_search/quick_sort_.a";
   end attributes;
end object;
```

This is the minimal definition which will attach the asset to the library. The asset's file must be copied into the appropriate directory in the library directory structure. For our example, the asset would be copied to `$RLF_LIBRARIES/Text/sort_and_search/quick_sort_.a`. Then when the LMDL specification had been retranslated, the asset would be visible from the reuse library.

When adding an asset, however, the library modeler should also describe the object representing the asset as fully as possible. This includes restricting and filling any relationships that may have been defined anywhere in the hierarchy directly above the object on a direct path to the root category. Also, any actions valid at the object or desired just at the object need to be defined, and any additional attributes which the objects has must also be defined.

Developing the most complete description of an asset may require the modeling of new categories to best describe the types of the objects inherited relationships. Filling relationships with other objects of the relationship's type is the best way to flesh out an object's description. It emphasizes that for this particular asset described by this object, that these particular things are true. It is the restriction and filling of an objects relationships that serve to differentiate it from other objects in the model. If an asset is not described in any way that distinguishes it from another similar asset, then the only way to discover the differences between the two assets would be to extract them both and examine them manually. This defeats the purpose of being able to model the differences directly with the library domain model.

The final definition of the new asset's object might look like this:

```
object "Example Quicksort" ( Quicksort ) is
   restricted relationships
      is_written_in (1 .. 1) of "Source Language";
      works_on (1 .. 1) of "Data Structure";
      has_worst_case_of (1 .. 1) of Quadratic;
      has_size_of (0 .. 1) of Number;
   end restricted;
   fillers
      Ada satisfies is_written_in;
      Array satisfies works_on;
      "N^2" satisfies has_worst_case_of;
      "Twenty-Four" satisfies has_size_of;
   end fillers;
   attributes
      file desc_source is "sort_and_search/exchange_sort_desc";
      file source is "sort_and_search/quick_sort_.a";
      string size_of is "24";
   end attributes;
   actions
      "View Code Size" is "Display Integer" on size_of;
      "View Source" is View on source;
      "Extract Source" is Extract on sour -
   end actions;
end object;
```

Removing Asset Descriptions from the Library Model

Sometimes assets will be removed from a reuse library either because they were outdated, have become invalidated, or were entered incorrectly. Remove the object definition describing the asset from the library domain model specification, and then retranslating, will serve to remove the asset from the library definition. If the object definition is likely to be reused by a new version of the asset it describes or by one with the same characteristics, it may only be necessary to uncouple the actual asset from its description by deleting the attributes which reference the actual asset. After retranslation, the object describing the asset can still be examined, but the actual asset cannot be extracted since it is no longer an attribute of the object.

10.2.2 Actions

While attributes are the way RLF attaches real data to the library domain model, actions are the way an RLF library user can manipulate a library's assets attached with RLF attributes. Because actions are what allows a library user to get at the library's reusable assets, they are of great concern to the library administrator. It may be the function of the library domain modeler to provide the action category definitions and attach the action to categories and actions in the main model, but the library administrator may wish to define the string attributes of the action categories to express how the action is actually performed. These strings often contain system and installation dependent information that the library administrator will need to provide. This sections describes how library model actions are modified by the library modeler in order to change the behavior of an RLF reuse library.

Adding New Actions

New actions are added by modifying the library model in two areas. One section of each RLF library model contains a sub-model rooted at the reserved category "`Action Definition`". "`Action Definition`" has two subcategories named `Action` and "`Action Type`". The sub-model rooted at `Action` contains descriptions of all the actions that can be available at other categories and objects in the library. The actions described in this sub-model are called "action categories." Although an action category can have other relationships and attributes which describe it, the most important parts of the action category are its `has_action_type` relationship which it inherits from `Action` and restricts locally and a string attribute which is used to invoke the action.

Below the "`Action Type`" category are sub-categories which describe the different types of actions available within the RLF reuse library. There are currently two types of actions supported, "`System String`" and "`Ada Procedure`". A "`System String`" type action uses the action category's string attribute as a string to be executed in the operating system shell. An "`Ada Procedure`" type action uses the action category's string attribute to match a built-in Ada procedure to call when the action is invoked. It is expected that additional types of RLF actions will be added in the future by adding additional subcategories to "`Action Type`". Possibilities include a message passing action and actions tailored to the environment in which the reuse library operates. New action types will interpret the action category's string attribute in the appropriate way for that type of action.

The category `Action` defines a relationship named `has_action_type` with a type of "`Action Type`". At each action category below `Action` this relationship is inherited and should be restricted to a more specific "`Action Type`". The following excerpt from the sort and search algorithms

library LMDL specification shows a part of the "Action Definition" sub-model including Action, "Action Type", and some of their subcategories.

```
category "Action Definition" ( Thing ) is
end category;

   category "Action Type" ( "Action Definition" ) is
   end category;

      category "System String" ( "Action Type" ) is
      end category;

      category "Ada Procedure" ("Action Type") is
      end category;

   category Action ( "Action Definition" ) is
      relationships
         has_action_type (1 .. 1) of "Action Type";
      end relationships;
   end category;

      category View ( Action ) is
         restricted relationships
            has_action_type of "System String";
         end restricted;
         attributes
            string is "xterm -e $RLF_PAGER ## &";
         end attributes;
      end category;
```

This example also shows the definition of a View action for the library. View is a "System String" type action which will have the string attribute "xterm -e $RLF_PAGER ## &" executed in an operating system shell when it is invoked. The restriction of has_action_type's type to "System String" is required so that RLF will know how to invoke the action correctly.

This example action definition also introduces substitution markers. When an action is of type "System String", the string attribute at the action category can contain special series of symbols which can be used to parameterize the action when it is invoked. A special marker, "##", in an action category's string attribute holds the place in the string where an argument to the action, called the "action target," will be substituted. The action target is supplied at the category where the action is available to be invoked. An action category's string attribute can also contain markers for "action agents" which are also supplied at the category or object where the action is available. Action agent substitution markers have the form "%%n", where n is a numeral from 1 to 9. For more information on action targets and agents, and how system string actions are invoked, consult the **RLF Modeler's Manual**.

RLF supports four built-in Ada procedure actions; Import, Export, Extract, and ''Display Attributes''. These actions are modeled by action categories in the action sub-model which have

restricted the has_action_type relationship to type "Ada Procedure". These action categories can be referenced from action definitions in the main part of the reuse library domain model. Any new actions of the "Ada Procedure" type which are Import or Export actions should probably be defined as privileged actions, since these operations are primarily library modeler operations and Import types actions may modify the reuse library model.

The first step to adding a new action to a library domain model is either to locate the desired action in the action sub-model section of the library's LMDL specification or to create the appropriate action category in the action sub-model. The same care should be taken in modeling action categories that is taken modeling parts of the main library model. Action categories which are subcategories of other action categories should be more specific forms of those categories. If the action being added is not related to any of the pre-existing example actions modeled with action categories, then a new action category describing the action should be defined as a direct subcategory of the category Action.

The new action category definition should restrict the has_action_type relationship inherited from Action to have type "System String" or "Ada Procedure". If the action type is "System String", then a string attribute should be defined at the action category which is the string to be executed in an operating system shell when the action is invoked. Substitution markers should be used in the string where action targets or agents will appear when the action is invoked. Action target, agents, and invocation is discussed in detail in the **RLF Modeler's Manual**. If the action type is "Ada Procedure", then the string attribute at the new action category should be one of "Import Asset", "Export Asset", "Extract Asset", or "Display Attributes" which are the built-in Ada procedure actions available.

An example of a new action category which will print a file associated with a category or object in the library model follows:

```
category Print ( Action ) is
-- this action category describes a general print action
   restricted relationships
      has_action_type (1 .. 1) of "System String";
   end restricted;
   attributes
      -- ## marks the file to be printed
      -- %%1 marks the UNIX print command to use
      -- %%2 marks any options to the print command
      -- also run the action in the UNIX background
      -- so the RLF application continues
      string print_command is "%%1 %%2 ## &";
   end attributes;
end category;
```

This LMDL fragment defines an action category named Print which describes a "System String" type action which prints its action target file using two action agents for the print command and any print command options. When an action which has Print as its action category is invoked, it will gather the required action target and agents from the category or object where it is invoked, process the action category's string attribute replacing the substitution markers with their actual

values, and then executing the final string in an operating system shell.

The other area of the library model which is modified to add new actions is the category definitions in the main library model. Actions are defined within a category very much like relationships, and are similarly available at subcategories and objects of the category or object where they are first defined. Once the action category is located or created in the action category sub-model, it can then be referenced at the categories where it will be available. Suppose the library modeler wants the library user to be able to print the source code of a quick sort implementation which is a reusable asset in the library. The print action could be defined at the object representing the quick sort implementation like this:

```
object "Example Quicksort" ( Quicksort ) is
   actions
      "Print Source" is Print on source with print_command, print_options;
   end actions;
end object;
```

This defines an action named "Print Source" at the object and tells RLF that the action is described by the action category named Print and will operate on the local file attribute named source. source is the action target. The action will also use the action agents print_command and print_options to modify the action invocation. If these attributes are defined like this:

```
object "Example Quicksort" ( Quicksort ) is
   attributes
      file source is "sort_and_search/quick_sort_.a";
      string print_command is "lpr";
      string print_options is "-Pprinter1";
   end attributes;
   actions
      "Print Source" is Print on source with print_command, print_options;
   end actions;
end object;
```

then, assuming the definition of the Print action category given above, when the action is invoked at the "Example Quicksort" object in the library, the file sort_and_search/quick_sort_.a will be printed using the *lpr* command on the printer specified in the option "-Pprinter1". (**NOTE:** The file name is relative to the Text/ subdirectory, which is a first-level subdirectory below the directory specified in the RLF_LIBRARIES environment variable.)

When defining new actions at categories in the main library model, it is useful to remember that actions can process a list of targets. For instance, suppose the library modeler wished to provide an action which would print the abstract, performance study, and source code for a particular quick sort implementation in the sort and search algorithms library. It would be best to use a list of targets so one action invocation by the user would print all the associated files. One solution in LMDL could look like this:

```
object "Example Quicksort" ( Quicksort ) is
   attributes
      file abstract is "sort_and_search/quick_sort_abstract";
      file performance_study is "sort_and_search/quick_sort_perf";
      file source is "sort_and_search/quick_sort_.a";
      string print_command is "lpr";
      string print_options is "-Pprinter1";
   end attributes;
   actions
      "Print All Data" is Print on abstract, performance_study, source
         with print_command, print_options;
      "Print Source" is Print on source with print_command, print_options;
   end actions;
end object;
```

This would provide a "Print Source" action to just print the implementation's source and a "Print All Data" action which would print the implementation's abstract, performance study, and source. When the "Print All Data" action is invoked, it will iterate over the list of targets performing the action described by action category Print for each file in the list. Again, more details on action invocation semantics and modeling appears in the **RLF Modeler's Manual**.

When new action categories have been added and new actions defined in the main library model which reference them, the library model definition must be retranslated by the library modeler using the LMDL translator, Lmdl, in order for the actions to be available from the RLF applications.

Modifying Actions

RLF "System String" type actions are modified by altering the action command string which is defined in the library model at the action's action category. This is a procedure usually done by the li The action category is a category in the action sub-model which is rooted at the reserved category Action. It describes the action and provides the action command string which is executed when the action is invoked. More information on the action sub-model, action categories, and action types can be found in the previous section on adding new actions.

The View action for the sort and search algorithms library is described at its action category as follows:

```
category View ( Action ) is
   restricted relationships
      has_action_type of "System String";
   end restricted;
   attributes
      string is "xterm -e $RLF_PAGER ## &";
   end attributes;
end category;
```

If the library modeler wanted to modify the view action so that it no longer ran in the UNIX

background and halted the RLF application instead, then the library model definition could be modified like so:

```
category View ( Action ) is
   restricted relationships
      has_action_type of "System String";
   end restricted;
   attributes
      string is "xterm -e $RLF_PAGER ##";
   end attributes;
end category;
```

Then when the library model had been retranslated using the LMDL translator, **Lmdl**, the view action, when invoked, would execute its new behavior and halt the RLF application until the view was complete.

Similarly, if the library modeler wished to change the **View** action so that it used a specific editor instead of using the **RLF_PAGER** environment variable to view the asset, then the view action category could be changed to this:

```
category View ( Action ) is
   restricted relationships
      has_action_type of "System String";
   end restricted;
   attributes
      string is "xterm -e /usr/ucb/view ##";
   end attributes;
end category;
```

Then when the library model had been retranslated using the LMDL translator, **Lmdl**, the view action, when invoked, would execute its new behavior and the asset would be viewed with *view* instead of the pager found in **RLF_PAGER**.

If the library modeler wished to do more complex operations when viewing an asset such as collecting metrics or doing configuration management, then the view action's operations could be put into a UNIX shell script, and the library model of the view action category modified as follows:

```
category View ( Action ) is
   restricted relationships
      has_action_type of "System String";
   end restricted;
   attributes
      string is "asset_view.csh ## &";
   end attributes;
end category;
```

Then when the library model had been retranslated using the LMDL translator, `Lmdl`, the view action, when invoked, would execute the *csh* shell script named `asset_view.csh` passing the name of the file to the script as a parameter. This script would also execute in the UNIX background allowing the RLF application to continue.

NOTE: To save time when modifying a library model's action categories, if the only changes made were changes to the string attributes at action categories, the LMDL translator should be invoked with the `-state` command line option when retranslating.

Removing Actions

Actions can be removed from the library model and thus the library in two ways. To make an action unavailable from certain categories or objects, but still present to others, the action definitions at the categories or objects can be removed. Since the action category for the action still exists in the action sub-model, it will still be available to categories and objects where the action has been kept. To remove the action from the library entirely, the action category description of the action should be removed from the action sub-model in the library model, and then all action definitions which reference that action's action category should be deleted. Both these methods for removing actions will only be evident after the library model specification had been retranslated by the LMDL translator, `Lmdl`.

10.2.3 Advice

RLF library advice is provided through the RLF's inferencing subsystem AdaTAU. Library advice is modeled in the Rule Base Definition Language (RBDL). More information on modeling library advice for RLF reuse libraries is provided section 9. This section addresses how the library modeler can manipulate advice attached to an RLF library.

Adding New Advice

Once new advice has been modeled and built, it can be attached to an RLF reuse library by editing the library's LMDL library model specification and then retranslating it. When making changes to the library model specification to change library advice, the LMDL translator, `Lmdl`, should be invoked with the `-state` command line option. This significantly reduces the amount of time that it takes to retranslate the library model.

Suppose the library modeler has produced a new advice module, or "inferencer," for a category or object in the library model and the library modeler wishes to make the inferencer available to the RLF applications. If the library modeler wishes to add the inferencer to the `Quicksort` category of the sort and search algorithms library, the following line would be added to the LMDL library model specification:

```
attach inferencer quicksort to Quicksort;
```

Now, if the inferencer has been named "quicksort" in its RBDL specification and has been built using the RBDL translator, `Rbdl`, then once the modified library model specification has been retranslated the library advice contained in the quicksort inferencer will be available at the category `Quicksort` from the RLF applications.

NOTE: The names of inferencers expressed in inferencer attachment LMDL clauses must appear in all lower case. This is because the RBDL translator currently converts all identifiers in the inferencer's RBDL specification to lower case. The identifier may appear in mixed case in the RBDL specification, but will be converted to lower case internally.

Modifying Advice

Library advice can be modified transparently by retranslating the RBDL specification that defines the inferencer containing the advice. No changes are necessary to the library's library model specification unless the name of the inferencer has changed. If the name has changed, the library model specification should be edited to reflect the name change and then retranslated to have the change installed. Running the LMDL translator, `Lmdl`, with the `-state` command line option is sufficient in this case.

Removing Advice

Library advice can be removed by removing the inferencer attachment in the library's library model specification and then retranslating the specification with the LMDL translator, `Lmdl`, and the `-state` command line option. For example, if library advice has been attached to the `Quicksort` category of the sort and search algorithms library with the following LMDL:

```
category Quicksort ( "Exchange Sorts" ) is
   restricted relationships
      has_best_case_of (1 .. 1) of Logarithmic;
      has_avg_case_of (1 .. 1) of Logarithmic;
      has_worst_case_of (1 .. 1) of Quadratic;
   end restricted;
   attributes
      file desc_source is "sort_and_search/exchange_sort_desc";
   end attributes;
end category;

attach inferencer quicksort to Quicksort;
```

then removing the attachment of the inferencer and leaving the `Quicksort` category definition like this:

```
category Quicksort ( "Exchange Sorts" ) is
   restricted relationships
      has_best_case_of (1 .. 1) of Logarithmic;
      has_avg_case_of (1 .. 1) of Logarithmic;
      has_worst_case_of (1 .. 1) of Quadratic;
   end restricted;
   attributes
      file desc_source is "sort_and_search/exchange_sort_desc";
   end attributes;
end category;
```

will remove the ability to access advice at this category once the library model specification has been retranslated. Also, since only a portion of the model defining library advice has been removed, it will be sufficient and quicker to retranslate the specification using the LMDL translator's `-state` command line option.

11 Modeling and Bug Support

11.1 What is a Bug

Library and advice domain modeling is a complex task, and the construction of consistent language encodings of domain models necessarily has many semantic constraints. Semantic checking in the LMDL and RBDL language translators is far from perfect, and certain tasks such as coordinating LMDL actions and attributes and testing RBDL inferencers still requires a lot of manual validation. Before reporting bugs in the RLF modeling software, it is important to make sure that the definition language specifications are consistent. It is especially important to ensure that occurrences of RBDL fact-attribute pairs that are supposed to be the same do indeed match. Bug reports should only be submitted when it is reasonably certain that the language specifications concerned are internally consistent. It is often useful to submit the specifications concerned with the bug report itself.

Receiving support with modeling in either LMDL or RBDL and checking consistency of definition language specifications is different than reporting RLF bugs. Bug reports are directed to their own electronic mail address. Requests for help modeling or debugging domain model specifications should be directed to the basic RLF electronic mail address. As with bug reports, it is most often useful to include the LMDL or RBDL specifications in question along with the request for help. If the specifications are very large, then it may be better to wait to send them until an RLF support person has responded to the initial request for help, and then mail them to that person directly. The appropriate electronic mail addresses for RLF are given in the following section.

11.2 Getting Help

This section describes how your RLF installation is supported. The installation of RLF includes a form called Program Problem Report (in file *Problem_Report*) that is used to identify any specific problems encountered in installing and using the software. The local RLF library administrator should know where to find this file and have more information about getting help. The **RLF Administrator's Manual** contains a more detailed description of RLF support.

There are two mailing lists established to help RLF modelers with their problems:

- **rlf@stars.rosslyn.paramax.com**
  This list provides a public forum for discussing RLF issues. Members of this list receive all messages sent to the list and may respond accordingly. Requests for help modeling with RLF or validating the consistency of definition language specifications should be sent to this list.
- **rlf-bugs@stars.rosslyn.paramax.com**
  Completed Program Problem Reports are sent to this address.

If errors in RLF applications or documentation are encountered, then a Program Problem Report should be filled out and sent by electronic mail to **rlf-bugs@stars.rosslyn.paramax.com**. If electronic mail is not available, the completed problem report should be mailed by standard post to:

RLF
Paramax STARS Center
12010 Sunrise Valley Drive
Reston, VA 22091

When the completed Program Problem Report is received, it will be acknowledged and the problem will be handled.

Requests for assistance with RLF are best handled through a Program Problem Report submitted to **rlf-bugs@stars.rosslyn.paramax.com** if RLF is having errors, or by electronic mail to the RLF mailing list, **rlf@stars.rosslyn.paramax.com**, if there is a lack of understanding or some questions. If electronic mail is unavailable, writing to the standard mail address in above will provide assistance.

A LMDL Syntax Summary

A.1 Notation

The syntax of the language is described using an extended BNF. The notation used is the same as the notation used throughout the Ada LRM. A brief description is given below. For a complete description see section 1.5 of the LRM.

lower_case_word
: nonterminal (e.g. library_model_spec).

*italicized_part*_lower_case_word
: refers to same nonterminal as the lower case word without italicized part. The italicized part is used to convey some semantic information (e.g. *category*_name).

bold_face_word
: language token (e.g. **category**).

`{item}`
: braces enclose item which may be repeated zero or more times.

`[item]`
: brackets enclose optional item.

`item1 | item2`
: alternation; either item1 or item2

Identifier	Defined Purpose
`"Action Definition"`	Root category of a model's action sub-model.
`Action`	Immediate child of `"Action Definition"`. Parent of all model action categories.
`"Action Type"`	Immediate child of `"Action Definition"`. Parent of all model action types.
`"System String"`	Immediate child of `"Action Type"`. Pre-defined library model action type.
`"Ada Procedure"`	Immediate child of `"Action Type"`. Pre-defined library model action type.
`has_action_type`	Relationship at an action category whose type is a child of `"Action Type"`.

Reserved Identifiers in LMDL

A.2 LMDL Syntax

library_model_spec ::=
library model *model*_name **is**
[incremental_indication]
[root_category]
{category_or_object_or_inferencer}
end model_name ;

incremental_indication ::=
extend library model *model*_name ;

category_or_object_or_inferencer ::=
category | object | inferencer

root_category ::=
root category *category*_name **is**
[local_relationships]
[differentiated_relationships]
[local_attributes]
[local_actions]
end root category ;

category ::=
category *category*_name (specializes) **is**
[local_relationships]
[restricted_relationships]
[differentiated_relationships]
[local_attributes]
[local_actions]
end category ;

specializes ::=
*category*_name {, *category*_name}

object ::=
object *object*_name (individuates) **is**
[restricted_relationships]
[differentiated_relationships]
[satisfied_relationships]
[local_attributes]
[local_actions]
end object ;

individuates ::=
*category*_name {, *category*_name}

local_relationships ::=
relationships
relationship {relationship}
end relationships ;

relationship ::=
*relationship*_name (number .. number_or_infinity)
of *category*_name ;

restricted_relationships ::=
restricted relationships
restriction {restriction}
end restricted ;

restriction ::=
range_restriction | value_restriction |
range_and_value_restriction

range_restriction ::=
*relationship*_name (number .. number_or_infinity) ;

value_restriction ::=
*relationship*_name **of** *category*_name ;

range_and_value_restriction ::=
*relationship*_name (number .. number_or_infinity)
of *category*_name ;

differentiated_relationships ::=
differentiated relationships
differentiation {differentiation}
end differentiated ;

differentiation ::= subset | partition

subset ::=
subset *relationship*_name **into**
relationship {relationship}
end subset ;

partition ::=
partition *relationship*_name **into**
relationship {relationship}
end partition ;

satisfied_relationships ::=
fillers
filler {filler}
end fillers ;

filler ::=
*object*_name **satisfies** *relationship*_name ;

local_actions ::=
actions
action {action}
end actions ;

action ::=
*action*_name **is** [**privileged**] *category*_name
[**on** targets] [**with** agents] ;

targets ::=
*target*_name {, *target*_name}

agents ::=
*agent*_name {, *agent*_name}

local_attributes ::=
attributes
attribute {attribute}
end attributes ;

attribute ::=
string_attribute |
file_attribute |
integer_attribute

string_attribute ::=
string [name] **is** string_literal ;

file_attribute ::=
file [name] **is** filename ;

integer_attribute ::=
integer [name] **is** number ;

inferencer ::=
attach inferencer *inferencer*_name
to *category_or_object*_name ;

*category_or_object*_name ::=
*category*_name | *object*_name

name ::= identifier | string_literal

filename ::= string_literal

identifier ::= letter {[underline] letter_or_digit}

letter_or_digit ::= letter | digit

letter ::= upper_case_letter | lower_case_letter

number_or_infinity ::= number | **infinity**

number ::= digit {digit}

B RBDL Syntax Summary

B.1 Notation

The syntax of the language is described using an extended BNF. The notation used is the same as the notation used throughout the Ada LRM. A brief description is given below. For a complete description see section 1.5 of the LRM.

lower_case_word
: nonterminal (e.g. inference_base_spec).

*italicized_part*_lower_case_word
: refers to same nonterminal as the lower case word without italicized part. The italicized part is used to convey some semantic information (e.g. *fact*_identifier).

bold_face_word
: language token (e.g. **fact**, ( ).

{item}
: braces enclose item which may be repeated zero or more times.

[item]
: brackets enclose optional item.

item1 | item2
: alternation; either item1 or item2

B.2 RBDL Syntax

inference_base_spec ::=
: factbase_schema_def
: [fact_parameters_def]
: initial_factbase_def
: {rule_base_definition}
: inferencer_def

factbase_schema_def ::=
: **fact base schema** *fact_base_schema*_identifier **is**
: fact_schema_def {fact_schema_def}
: **end** [*fact_base_schema*_identifier] ;

fact_schema_def ::=
: attribute_name_list : attribute_type
: [attribute_value_list] ;

attribute_name_list ::= attribute_name {, attribute_name}

attribute_type ::= **some_of** | **one_of** | **any** | **reference**

attribute_value_list ::= (attribute_value {, attribute_value})

attribute_name ::= *fact*_identifier | string_literal

attribute_value ::= identifier | string_literal | number

fact_parameters_def ::=
fact parameters is
[import_list] [export_list]
end fact parameters ;

import_list ::= **imports** : (param_description_list) ;

export_list ::= **exports** : (param_description_list) ;

param_description_list ::=
param_description {, param_description}

param_description ::=
*fact*_identifier => **optional** |
*fact*_identifier => **mandatory** |
*fact*_identifier => **focal**

initial_factbase_def ::=
initial fact base *initial_fact_base*_identifier **is**
fact_list ;
end [*initial_fact_base*_identifier] ;

fact_list ::= **null** | fact_def {, fact_def}

fact_def ::= (attribute_name , attribute_value)

neg_fact_def ::= ~ fact_def

rule_base_definition ::=
IRule_base_def |
question_base_def |
QRule_base_def |
FRule_base_def

IRule_base_def ::=
irule base *IRule_base*_identifier **is**
IRule_def {IRule_def}
end [*IRule_base*_identifier] ;

IRule_def ::=
irule *IRule*_identifier **is**
antecedent : antecedent_fact_list ;
consequent : consequent_fact_list ;
[justification]
end irule ;

antecedent_fact_list ::=
fact_def {, fact_def}

consequent_fact_list ::=
fact_def |
neg_fact_def |
consequent_fact_list , fact_def |
consequent_fact_list , neg_fact_def

justification_def ::=
justification : text_block ;

question_base_def ::=
question base *question_base*_identifier **is**
question_def {question_def}
end [*question_base*_identifier] ;

question_def ::=
question *question*_identifier **is**
text : text_block ;
type : attribute_type ;
[possible_responses]
end question ;

possible_responses ::=
responses : response_list

response_list ::= response {response}

response ::=
response_display {| response_display} =
consequent_fact_list ;

response_display ::= string_literal

QRule_base_def ::=
qrule base *QRule_base*_identifier
(*question_base*_identifier) **is**
QRule_def {QRule_def}
end [*QRule_base*_identifier] ;

QRule_def ::=
qrule *QRule*_identifier **is**
antecedent : antecedent_fact_list ;
question : *question*_identifier ;
weight : rule_weight ;
[justification]
end qrule ;

rule_weight ::= number

FRule_base_def ::=
frule base *FRule_base*_identifier **is**
FRule_def {FRule_def}
end [*FRule_base*_identifier] ;

FRule_def ::=
frule *FRule*_identifier **is**
antecedent : antecedent_fact_list ;
export : export_fact_list ;
focus : *inferencer*_identifier ;
weight : rule_weight ;
[justification]
end frule ;

export_fact_list ::= fact_list

inferencer_def ::=
inferencer *inferencer*_identifier **is**
[IRule_base_specification]
[QRule_base_specification]
[FRule_base_specification]
end [*inferencer*_identifier] ;

IRule_base_specification ::= **irule base** : *IRule_base*_identifier ;

QRule_base_specification ::= **qrule base** : *QRule_base*_identifier ;

FRule_base_specification ::= **frule base** : *FRule_base*_identifier ;

identifier ::= letter {[underline] letter_or_digit}

letter_or_digit ::= letter | digit

letter ::= upper_case_letter | lower_case_letter

number ::= digit {digit}

text_block ::= { any_letter_but_bracket {any_letter_but_bracket} }

C Starter Library Model Template

This appendix presents a LMDL specification appropriate for using as a template in new library model development. This specification includes some top-level organization and an example action sub-model. Every library model which defines actions which will be invoked from the **Graphical_Browser** or **Library_Manager** must include an action sub-model. The example library models included with RLF all contain a subset of the example action sub-model in this template. The minimum requirements of the action sub-model for a library model are given in section 3.3.6.

```
--|
--| This LMDL library model can be used as a template to begin development
--| of new library model specifications.  It includes a developed, example
--| action sub-model and some basic top-level organization which can help
--| keep the library model cogently organized.
--|

library model "Library Model Template" is

   --| "Library Model Template" can be substituted with the desired name
   --| of the library which will appear in the menus of the RLF tools.

   root category Thing is
      --| The root category should be the most general class of things
      --| described in the library model.  All subcategories should be
      --| subsets of the set of things described by the root category.
      --| This is the traditional name for the root category.
   end root category;

   category "Main Sub-Model" (Thing) is
      --| This category can be the root of the main sub-model of this
      --| specification which describes all the reusable assets made
      --| available in the reuse library and modeled in this specification.
      --| The name of this category should probably be changed to better
      --| represent the contents of the library model.
   end category;

   category "Relationship Values" (Thing) is
      --| This category can be the root of the sub-model which defines
      --| categories and objects which serve as filler types and values
      --| of relationships describing categories and objects in the
      --| main sub-model.  This category is probably appropriately named.
   end category;

   category "Action Definition" (Thing) is
      --| This category is the root of the action sub-model which must exist
      --| in order for the RLF reuse library constructed from this
      --| specification to be able to invoke actions described here.
```

```
      --| THE NAME OF THIS CATEGORY CANNOT BE CHANGED, although relationships
      --| may be added and it may appear once as the child of any appropriate
      --| category in the library model.  This is probably the best location
      --| for this category, however.
   end category;

      category "Action Type" ("Action Definition") is
         --| THE NAME OF THIS CATEGORY CANNOT BE CHANGED.
      end category;

         category "System String" ("Action Type") is
            --| THE NAME OF THIS CATEGORY CANNOT BE CHANGED.
         end category;

         category "Ada Procedure" ("Action Type") is
            --| THE NAME OF THIS CATEGORY CANNOT BE CHANGED.
         end category;

         category "Message Pass" ("Action Type") is
         end category;

      category Action ("Action Definition") is
         --| THE NAME OF THIS CATEGORY AND THE RELATIONSHIP DEFINED
         --| HERE CANNOT BE CHANGED.
         relationships
            has_action_type (1 .. 1) of "Action Type";
         end relationships;
      end category;

         --|
         --| The following action definitions describe the existing
         --| RLF built-in "Ada Procedure" actions and some example
         --| "System String" actions for use in the example libraries
         --| and in new library models.
         --|

         category View (Action) is
            restricted relationships
               has_action_type of "System String";
            end restricted;
            attributes
               string is "xterm -e $RLF_PAGER ## &";
            end attributes;
         end category;

         category Edit (Action) is
            restricted relationships
               has_action_type of "System String";
```

```
   end restricted;
   attributes
      string is "xterm -e $RLF_EDITOR ## &";
   end attributes;
end category;

category Mail (Action) is
   restricted relationships
      has_action_type of "System String";
   end restricted;
   attributes
string is "Mail -s 'message ##' $RLF_ADMIN < /dev/null > /dev/null &";
   end attributes;
end category;

category Extract (Action) is
   restricted relationships
      has_action_type of "Ada Procedure";
   end restricted;
   attributes
      string is "Extract Asset";
   end attributes;
end category;

category Import (Action) is
   restricted relationships
      has_action_type of "Ada Procedure";
   end restricted;
   attributes
      string is "Import Asset";
   end attributes;
end category;

category Export (Action) is
   restricted relationships
      has_action_type of "Ada Procedure";
   end restricted;
   attributes
      string is "Export Asset";
   end attributes;
end category;

category "Display Attributes" (Action) is
   restricted relationships
      has_action_type of "Ada Procedure";
   end restricted;
   attributes
      string is "Display Attributes";
```

```
        end attributes;
    end category;


end "Library Model Template";
```

D .rlfrc Start-Up File Syntax Summary

D.1 Notation

The syntax of the language is described using an extended BNF. The notation used is the same as the notation used throughout the Ada LRM. A brief description is given below. For a complete description see section 1.5 of the LRM.

lower_case_word
: nonterminal (e.g. library_model_spec).

*italicized_part*_lower_case_word
: refers to same nonterminal as the lower case word without italicized part. The italicized part is used to convey some semantic information (e.g. *category*_name).

bold_face_word
: language token (e.g. **category**).

`{item}`
: braces enclose item which may be repeated zero or more times.

`[item]`
: brackets enclose optional item.

`item1 | item2`
: alternation; either item1 or item2

D.2 .rlfrc File Syntax

```
startup_file ::=
    {setting}
```

setting ::=
default_directory |
default_library |
start_category |
view_type |
view_depth |
topology_flag |
cardinality_flag |
layout_offset |
bitmap |
tau_setting |
debug_flag |
working_directory |
history_list_length |
default_editor |
default_pager |
translator_setting

default_directory ::=
library directory : pathname

default_library ::=
library : name

start_category ::=
initial category : name

view_type ::=
view type : agg_or_spec

agg_or_spec ::=
relationship | specialization

view_depth ::=
view depth : [agg_or_spec :] depth_setting

depth_setting ::=
all | integer

topology_flag ::=
topology : flag_setting

flag_setting ::=
yes | no | true | false | on | off

cardinality_flag ::=
cardinality : flag_setting

layout_offset ::=
layout offset : [x_or_y :] integer

x_or_y ::=
x | **y**

bitmap ::=
node bitmap : category_or_object
[: has_attribute {has_attribute}] : pathname

category_or_object ::=
category | **object**

has_attribute ::=
inferencer | **actions** | **attributes**

tau_setting ::=
advice : tau_setting_type

tau_setting_type ::=
explanations : explanation_type |
automatic move : flag_setting

explanation_type ::=
none | **all** | explanation_kind {explanation_kind}

explanation_kind ::=
reasoning | **questions** | **moving**

debug_flag ::=
debug : flag_setting

working_directory ::=
working directory : pathname

history_list_length ::=
history length : integer

default_editor ::=
editor : string

default_pager ::=
pager : string

translator_setting ::=
translator : translator_type

translator_type ::=
lmdl : lmdl_setting |
rbdl : rbdl_setting

lmdl_setting ::=
quiet_translation | translate_only | default_input_spec

rbdl_setting ::=
quiet_translation | default_input_spec

quiet_translation ::=
quiet : flag_setting

translate_only ::=
only : model_or_state

model_or_state ::=
model | state

default_input_spec ::=
default specification : pathname

integer ::= digit {digit}

name ::= identifier | " character {character} "

identifier ::= letter {[underline] letter_or_digit}

letter_or_digit ::= letter | digit

letter ::= upper_case_letter | lower_case_letter

pathname ::= printable_non_whitespace {printable_non_whitespace}

D.3 Example .rlfrc File

```
--|
--| Sample startup file for the Reuse Library Framework version 4.0
--|

--|
--| Library directory or name specifications
--|
--library directory : /path/Libraries
--library : "Sort and Search Algorithms"
```

```
--|
--| Parameters for the RLF Graphical Browser
--|
topology : off
cardinality : off
layout offset : x : 20
layout offset : y : 5
history length : 50
view type : specialization
view depth : relationship : 2

--|
--| AdaTau inferencing settings
--|
advice : explanations : all
advice : automatic move : false

--|
--| Bitmaps for nodes
--|
--node bitmap : category : /path/box_m.xbm
--node bitmap : category : inferencer : /path/box_I_m.xbm
--node bitmap : category : actions : /path/box_A_m.xbm
--node bitmap : category : inferencer actions : /path/box_AI_m.xbm
--node bitmap : object : /path/cube_m.xbm
--node bitmap : object : inferencer : /path/cube_I_m.xbm
--node bitmap : object : actions : /path/cube_A_m.xbm
--node bitmap : object : inferencer actions : /path/cube_AI_m.xbm

--|
--| Specification translator settings
--|
translator: Lmdl: quiet: no
translator: Rbdl: quiet: no
```

E PCTE and RLF

In most respects, the PCTE version of this delivery of RLF will operate in the same manner as the UNIX version. This appendix, however, will present the differences in the PCTE and UNIX versions of RLF and present some conventions which can be used to produce library models which will be portable between versions. It will also list some requirements of the PCTE version which are not UNIX requirements. This appendix assumes knowledge of PCTE, the Emeraude PCTE product, and the *esh* shell.

E.1 File Naming Restrictions

The Emeraude implementation of PCTE places restrictions on the length of object names and makes assumptions about the use of '.' in object names. The names of files containing assets which are available in an RLF reuse library are restricted to 32 characters in length when using PCTE. These are the files that reside beneath the `Text/` subdirectory of any directory where RLF libraries have been constructed. Additionally, the names of files containing reusable assets in the library should not contain the '.' character, since this indicates a special meaning to the Emeraude implementation of PCTE. The convention established by this version of RLF for PCTE is to replace any '.' characters in file names with the underscore character, '_'. An exception to this convention is the `.rlfrc` start-up file, which the PCTE version of RLF will look for as an entity named `rlfrc.e`.

To increase the similarity in the way libraries are represented in the UNIX and PCTE versions, and to ease transition between versions, the preferred link type of every object in or beneath the directory object where the library was built must be set to ".e". This includes files representing a library's assets and any action scripts which might appear below the `Text/` directory. The preferred link type of the directory object indicated by the environment variable, `RLF_LIBRARIES`, also needs to be ".e" so that its subdirectories can be traversed easily.

Library representations built with the PCTE version of RLF also require a directory object named `rlf_tools` to be a first-level subdirectory of the directory object where the library is built. This directory object must also contain two tools named `ascii_file.tool` and `displ_attr.tool`. These tools are required for RLF's default actions to operate correctly.

For excellent examples of library model construction for the PCTE version of RLF, examine the `.esh` versions of the build scripts for the example libraries delivered with RLF. These scripts are found in each subdirectory of the `models/` directory of an RLF installation. These scripts can be modified and reused to help automate the procedures required to build an RLF reuse library with the PCTE version.

E.2 Action Modeling with PCTE

The modeling of "`System String`" type actions in PCTE has stricter requirements than for UNIX. (See section 8.2 for details on UNIX action modeling.) In the UNIX version, commands may be placed directly in the string attribute of the action category, and then this command will be executed in its own UNIX shell when an action which references it is invoked. PCTE, however, must invoke an *esh* process to perform actions. Because of this, all "`System String`" type actions in the PCTE version of RLF must be in an *esh* script which can be executed in a PCTE process. This

is very similar to encapsulating an action in a *csh* script which is executed by UNIX. Additional parameters may be supplied just like for any other *esh* script, but no pipes, output redirection, backgrounding of the task, or multiple commands (colon separated list of commands) are allowed in the string attribute which represents the action at the action category. If any of these capabilities are needed for the action to meet its goals, then these things should be done within the *esh* script which performs the action. Also, within the script which performs the action, if it is necessary to execute UNIX-only, non-encapsulated programs (e.g. **xloadimage**), then the script will have to use the *esh* command **epath** to convert a PCTE pathname to a UNIX file name.

The PCTE version of RLF assumes these *esh* action scripts can be found in the **Text/** subdirectory of the directory object where the libraries were constructed. (This is the directory usually specified using the **RLF_LIBRARIES** environment variable.) The modeler must specify any additional paths in the LMDL specification. Typically, the scripts are deposited in a model-specific directory object in the **Text/** directory object. For example, if a library describing animals has an action which invoked an *xterm* and ran *less* in it to view an asset, the final location of the *esh* action script might be

```
$RLF_LIBRARIES/Text/animals/xterm_less.tool
```

Scripts must be of type **sctx** and should use the link extension **.tool** in the PCTE object base. The script writer and installer should verify that the execute permissions are set for any action scripts. The command "**obj_set_mode a+x <pathname>**" executed in *esh* would do this for the script indicated by **<pathname>**. If the script is created as a UNIX file and has the correct permissions in UNIX, the PCTE copy will have the correct permissions.

For a larger example, suppose an RLF library model is being developed which will be a repository of bitmaps. A necessary action for this library is one which allows the user to view a bitmap which is a candidate for reuse. A "**View Bitmap**" action category which might appear like this in the action sub-model in the UNIX version:

```
category "View Bitmap" (Action) is
   restricted relationships
      has_action_type of "System String";
   end restricted;
   attributes
      string is "xloadimage ## &";
   end attributes;
end category;
```

In the PCTE version, this action should be modeled like this:

```
category "View Bitmap" (Action) is
   restricted relationships
      has_action_type of "System String";
   end restricted;
   attributes
```

```
        string is "bitmaps/xloadimage.tool ##";
    end attributes;
end category;
```

In this example, **xloadimage.tool** is an *esh* script which PCTE will invoke in a separate process. The contents of **xloadimage.tool** might look like this:

```
xloadimage "`epath $1`" &
```

This example illustrates a few of the differences between the UNIX and PCTE versions of RLF. It shows the necessary encapsulation of an action in an *esh* script for PCTE which appears as the action category's string attribute with no pipes, file re-direction, multiple commands, or backgrounding. It also shows that once inside the script, these things can be done as usual, and that UNIX-only tools need to use **epath** to resolve the actual UNIX pathname of a PCTE object with contents. Using the action script location convention, this script would be located in

```
$RLF_LIBRARIES/Text/bitmaps/xloadimage.tool
```

where **RLF_LIBRARIES** indicates the directory object where the library was constructed and **bitmaps/** is a model-specific subdirectory of **Text/** where files related to the bitmaps library will reside.

This example also shows how UNIX/PCTE portable library models can be developed if useful. The PCTE version of the "**View Bitmap**" action above would work equally well for UNIX if a *csh* script named **xloadimage.tool** was written with the following contents:

```
#! /bin/csh -f
xloadimage $* &
```

If appropriate versions of **xloadimage.tool** were installed in the library directories according to which version of RLF was being used, and file names for assets and related files were carefully chosen (according to the guidelines above) for the UNIX version, then the production of UNIX/PCTE portable library model specifications is not difficult.

F The SNDL to LMDL Translator

The library domain modeling language for RLF 4.0 has been changed from the Semantic Network Definition Language (SNDL) used in previous versions of RLF to the new Library Model Definition Language (LMDL). To make the transition between languages easier, and to enable the reuse of existing SNDL specifications with RLF 4.0, a translator which will translate SNDL to LMDL has been included with RLF 4.0. The translator, `Sndl_to_Lmdl`, is included in the `bin/` directory of the RLF 4.0 release. If `Sndl_to_Lmdl` is in the user's path and the following command is issued in the operating system shell:

```
Sndl_to_Lmdl -o spec.lmdl spec.sndl
```

then, if `spec.sndl` is an existing semantic network specification written in SNDL, the LMDL equivalent of the specification will be written into the file named `spec.lmdl`. Any problems that the translator experienced will be printed out. This information can be logged using options described below.

A description of the options available to the SNDL to LMDL translator follows:

Synopsis

```
Sndl_to_Lmdl [-help] [<filename>] [-pc] [-indent <num>]
             [-o <name>] [-l <name>]
```

Available command line arguments:

```
-help          prints available command line arguments.

<filename>     name of file to translate, if an input file is
               not specified then input defaults to stdin.

-pc            Preserve the case of all names (e.g. concept names,
               individual names, and role names) that occur in the
               input SNDL specification.  If this option is not
               specified, all names are converted to lower case and
               the resultant network generated by the LMDL specification
               will be identical to the network generated by the
               SNDL specification.

               This option is provided because LMDL supports
               case sensitive identifiers as well as arbitrary strings
               for names, where as SNDL permits mixed case identifiers
               for names in the specification, but SNDL automatically
               converts them to lower case in the generated network.

-indent <num>  Indent each major language construct by <num> spaces,
               where <num> a non-negative integer.  If this option is
```

not specified, an indentation of 3 spaces is used.

-o <name> Write the generated LMDL specification to a file with the specified name instead of stdout.

-l <name> Write the log file to a file with the specified name instead of stderr. The log file gives status information about the translation. Status information includes an indication of whether the translation was successful, a list of comments not transferred from the SNDL to the LMDL specification, and any errors detected in processing the input specification.

References

[Ada83] United States Department of Defense. *Reference Manual for the Ada Programming Language*, February 1983.

[BS85] R. Brachman and J. Schmolze. An Overview of the KL-ONE Knowledge Representation System. *Cognitive Science*, 9(2):171–2116, Spring 1985.

[FHM+83] M. W. Freeman, L. Hirschman, D. P. McKay, F. L. Miller, and D. P. Sidhu. Logic Programming Applied to Knowledge-Based Systems, Modelling, and Simulation. In *Proceedings of the Conference on Artificial Intelligence*, pages 177–193, April 1983.

[KBR86] T. S. Kaczmarek, R. Bates, and G. Robins. Recent Developments in NIKL. In *Proceedings AAAI-86*, pages 978–985, Philadelphia, PA, August 1986. Fifth National Conference on Artificial Intelligence.

[PKP+82] Teri F. Payton, S. E. Keller, John A. Perkins, S. Rowan, and Susan P. Mardinly. SSAGS: A Syntax and Semantics Analysis and Generation System. In *Proceedings of COMPSAC '82*, pages 424–433, 1982.